# D10

EDMRST_Finish                     10   (RSTinitfin.c)
EDMRST_Initialize..........3   (RSTinitfin.c)
EDMRST_Ping                        9   (RSTinitfin.c)

```
 2  /***********************************************************
 3  **
 4  ** File Name:     RSTinitfin.c
 5  **
 6  ** Copyright (c) 1998,1999 by EMC Corporation.
 7  **
 8  ** Purpose:
 9  **          This module contains the Restore API functions to
10  **          initialize and terminate the restore operation.
11  **
12  ** Table of Contents:
13  **          ----------------
14  **          API Functions:
15  **                 EDMRST_Initialize
16  **                 EDMRST_Finish
17  **
18  **          Internal Functions:
19  **
20  **
21  **          Compile-Time Options:
22  **                 This section must list any compile time definitions
23  **                 which will affect this header.
24  **
25  ** ***********************************************************/

28  /* The following provides an RCS id in the binary that can be located
29  ** with the what(1) utility.  The intent is to keep this short.
30  */

32  #ifndef lint
33  static char RCS_id [] = "$RCSfile$ "
34                          "$Revision$ "
35                          "$Date$" ;
36  #endif

39  /*
40   * Feature test switches.
41   *
42   * Standard defines required to turn on OS features go here.
43   *
44   * The following is required for code that uses POSIX API's.
45   * Remove for non-POSIX, non-portable code.
46   */
47  #define _POSIX_SOURCE 1

49  /*
50   * System headers.
51   */
52  #include <pwd.h>

55  /*
56   * Epoch headers.
57   */
58  #include <eb/eb_port.h>
59  #include <eb/rb_log.h>

61  /*
62   * Local headers.
63   */
64  #include <RSTinterns.h>
```

```
65  #include <RSTsup_csm.h>

67  /*
68   * Comms headers.
69   */
70  #include <restore/csc_EDMDispatch.h>
71  #include <restore/csc_EDMRestoreEng.h>
72  #include <restore/dispatch_daemon.h>
73  #include <restore/restore_engine.h>
74  #include <edmlink/edmlink_api.h>

76  /*
77   * #defines, structures, typedefs local to this source file
78   */

81  /*
82   * Global declarations
83   */

85  internalHandlePtr handlePtr = NULL;
```

```c
 87   /***************************************************************
 88   *
 89   * EDMRST_Initialize:
 90   *
 91   * This function takes care of all the initialization for a recovery
 92   * session. This must be called prior to any of the other functions
 93   * in the Recover API.
 94   *
 95   * Parameters:
 96   *
 97   *    hostname      (I) - The machine name of the server to use.
 98   *    svrHdl
 99   *                  O) - A handle to receive a pointer to this user's client
100   *                       handle for the Restore Engine connection.
101   *
102   *    timeout       (I) - The maximum number of seconds to wait for the connection
      *                       to the Restore Engine process to be completed.
      *
      ***************************************************************/
104   eerrno_ty
105   EDMRST_Initialize( hostname_ty     hostname,
106                      serverHandle    *svrHdl,
107                      unsigned long   timeout )
108   {
109       eerrno_ty api_status = E_SUCCESS;

111       uid_t    human_uid;
112       struct   passwd *pw;
113       char     *human_uidname ;

115       RE_initialize_args      re_init_args;
116       RE_status_result        *re_init_result;
117       rpc_if_handle_t         re_handle;
118       rpc_binding_handle_t    re_if_spec;
119       int                     retval;
120       time_t                  end_time;

122   #ifdef DEBUG
123   #define RPC_TIMEOUT     3600
124       struct timeval  rpc_timeout;
125   #endif

127   /*************** BEGINNING OF Dispatch Daemon STUFF ***************/
128       error_status_t status;
129       DD_initialize_args        initargs;
130       DD_getservicestatus_args  statargs;
131       DD_initialize_result      *initres = NULL;
132       DD_getservicestatus_result  *statres = NULL;
133       rpc_if_handle_t    if_spec;

136       time( &end_time );      /* compute time to give up waiting */
137       end_time += timeout;

139       memset(&if_spec,0,sizeof(rpc_if_handle_t));
140       memset(&re_if_spec,0,sizeof(rpc_if_handle_t));

142       if (svrHdl == NULL || hostname == NULL )
143       {
144           return( EP_RB_RECOVER_BAD_ARGS ) ;
145       }

147       rec_api_log_begin( "edmrestore_api" );
                            /* init logs, ignore errs?? */
```

```c
149       /* get user name to pass to DD and RE */
150       human_uid = getuid();
151       pw = getpwuid( human_uid );
152       if (pw == NULL || NULL == pw->pw_name )
153       {
154           /* Trouble. */

156           rec_api_log_csm(SUB_CSM_USER_NOT_IN_PASSWD, NULL);
157           return(EP_RB_RECOVER_PERMISSION_DENIED);
158       }

160       human_uidname = pw->pw_name;

162       handlePtr = (internalHandle *) calloc(1, sizeof(internalHandle));

164       /* Use this macro to setup the interface spec */
165       CLIENT_IFSPEC(if_spec);

167       /*
168       ** Arrive at a server binding.   Note that if they didn't give us
169       ** a valid host parameter, this will fail and drop through and
170       ** return NULL in the end.
171       ** This call will get and store a fully resolved binding
172       ** handle to the host.  The first time we ever call the host,
173       ** csc_get_handle will resolve and store the binding.  If we
174       ** ever use csc_get_handle to talk to the same host again,
175       ** it will just give back the previously resolved binding.
176       */

177       retval = csc_get_handle((unsigned char *) hostname,
178                                if_spec,
179                                SERVER_GROUP,
180                                &handlePtr -> dd_binding_handle,
181                                &status);

183       /*
184       ** Find out if we got csc handle and see if status is bad.
185       ** error_status_ok is a macro defined in cscomm.h.
186       */

188       if ((status != error_status_ok) || (retval == 0))
189       {
190           /* If errno not set, use status if it is a valid errno value */

192           if ( errno == 0 )
193               errno = ( strerror( status ) ? status : ETIME );

195           rec_api_log_csm( SUB_CSM_RPC_FAIL,
196                 "failure finding edmdispd to start restore engine" );

198           return EP_RB_RECOVER_SERVERFAIL;
199       }

201       errno = 0;

203   #ifdef DEBUG
204       /* Increase rpc timeout during debugging */
205       rpc_timeout.tv_sec = RPC_TIMEOUT;
206       rpc_timeout.tv_usec = 0;
207       clnt_control( handlePtr->dd_binding_handle, CLSET_TIMEOUT,
208                     (char *)&rpc_timeout );
209   #endif

211       initargs.service = DD_SERVICE_RESTORE;
212       initargs.hostname = hostname;
213       initargs.username = human_uidname;
```

```
214  1    initargs.timeout = timeout;

216  1    initres = dd_initialize_1(
                &initargs, handlePtr -> dd_binding_handle );
217  1                /* Will have _1 for RPC call */

219  1    if (initres == NULL)
220  2    {
221  2        return EP_RB_RECOVER_RPC_FAIL;
222  1    }

224  1    statargs.service_handle = initres -> service_handle;
225  1    statargs.status = 0;

227  1    statres = dd_getservicestatus_1(
                &statargs, handlePtr->dd_binding_handle );

229  1    if (statres == NULL)
230  2    {
231  2        return EP_RB_RECOVER_RPC_FAIL;
232  1    }

234  1    while (statres -> status == DD_SERVICE_STARTING )
235  2    {
236  2        time_t  now;

238  2        xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
239  2        time( &now );
240  2        if (now >= end_time)
241  3        {
242  3            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                    "timeout waiting for edmdispd to start restore
                    engine" );
243  3
244  3            return EP_RB_RECOVER_SERVERFAIL;
245  2        }

247  2        sleep(1);

249  2        statres = dd_getservicestatus_1( &statargs,
250  2                    handlePtr ->
                          dd_binding_handle );

252  2        if (statres == NULL)
253  3        {
254  3            rec_api_log_csm( SUB_CSM_RPC_FAIL,
255  3                "failure getting status from edmdispd while starting restore
                    engine" );
256  3            return EP_RB_RECOVER_RPC_FAIL;
257  2        }
258  1    }

260  1    if (statres -> status != DD_SERVICE_RUNNING)
261  2    {
262  2        rec_api_log_csm( SUB_CSM_RPC_FAIL,
263  2            "edmdispd failure while starting restore
                engine" );
264  2        xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
265  2        return EP_RB_RECOVER_SERVERFAIL;
266  1    }

268  1    memcpy( handlePtr -> opaque128,
269  1            statres -> handle.handle_val,
270  1            sizeof(handlePtr -> opaque128) );

272  1    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
```

```
274  1    /* ************************** END of Dispatch Daemon STUFF ************************** */

276  1    /* Restore Engine FUNCTIONALITY BEGINS HERE */

276  1    /*      RE_CLIENT_IFSPEC(re_if_spec); */

276  1    retval = csc_private_ifspec_init(
                    (unsigned char *) handlePtr -> opaque128,
                    RE_PROGNUM,
                    RE_VERSNUM,
                    &re_if_spec,
                    &status);

285  1    if (retval == 0)
288  2    {
289  2        rec_api_log_csm( SUB_CSM_RPC_FAIL,
290  2            "failure initializing interface to restore
291  2            engine" );
292  1
294  1        return EP_RB_RECOVER_SERVERFAIL;
295  2    }

296  2    api_status = EP_RB_RECOVER_SERVERFAIL;
297  2    do {
298  2        time_t  now;
299  2        time( &now );
300  3        if (now >= end_time)
301  3        {
302  3            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                    "timeout connecting to restore
                    engine" );

303  2            return EP_RB_RECOVER_SERVERFAIL;
305  2        }

306  2        sleep( 1 );     /* give restore engine time to get going */
307  2        retval = csc_connect_to_rpc_service(
308  2                    (unsigned char *)hostname,
309  2                    re_if_spec,
310  2                    RE_CLIENT_GROUP,
311  2                    &handlePtr ->
                          re_binding_handle,
312  2                    &status);
313  2        if ((status == error_status_ok) && (retval != 0))
314  1            api_status = E_SUCCESS;
316  1    } while (api_status != E_SUCCESS);

317  2    if (api_status == E_SUCCESS)
318  2    {
319  2        re_handle = handlePtr -> re_binding_handle;

320  2    #ifdef DEBUG
321  2    /*      increase rpc timeout during debugging */
322  2        rpc_timeout.tv_sec = RPC_TIMEOUT;
323  2        rpc_timeout.tv_usec = 0;
324  2        clnt_control( re_handle, CLSET_TIMEOUT, (
                        char *)&rpc_timeout );
325  2    #endif

326  2        re_init_args.username = human_uidname;
327  2        set_rpc_obj( re_initialize, &re_init_args.RPCobjID );
328  3        re_init_result = re_initialize_1(
                        &re_init_args, re_handle );
329  3        if (!re_init_result) {
330  3            api_status = EP_RB_RECOVER_RPC_FAIL;
331  3            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                    "failure communicating with restore
```

```
332  2        }
333  3            else {
334  3                api_status = re_init_result->status;
335  3                /* release RPC result struct */
336  3                xdr_free( xdr_RE_status_result, (
                          char *)re_init_result);
337  2            }
338  1        }
340  1            else
341  1                rec_api_log_csm( SUB_CSM_RPC_FAIL,
342  1                    "failure connecting to restore engine" );
344  1        if (
          api_status == E_SUCCESS)     /* return rest eng handle on success */
345  1            *svrHdl = (serverHandle)re_handle;
347  1        return( api_status );
349  }
          /* End of EDMRST_Initialize() */
```

```
351  /***************************************************************
352   *
353   * Ping:
354   *
355   * This function allows a ping to be issued in order to keep the
356   * engine alive and running so that the engine will not time out.
357   *
358   * Parameters:
359   *
360   * svrHdl (I) - A pointer to this user's client handle for the
361   *              Restore Engine (server) connection.
362   *
363   ***************************************************************/
364  eerrno_ty    EDMRST_Ping( serverHandle svrHdl )
365  {
366      eerrno_ty         api_status = E_SUCCESS;
367      RE_null_args      re_ping_args;
         RE_status_result  *re_ping_result = NULL;

370      if ( NULL == svrHdl || NULL == handlePtr
371           || svrHdl != handlePtr->re_binding_handle )
372      {
373          return( EP_RB_RECOVER_BAD_ARGS );
374      }

376      set_rpc_obj( re_ping, &re_ping_args.RPCobjID );
377      re_ping_result = re_ping_1( &re_ping_args, svrHdl );
378      if (NULL == re_ping_result) {
379          api_status = EP_RB_RECOVER_RPC_FAIL;
380          rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL );
381      }
382      else {
383          api_status = re_ping_result->status;
384          /* release RPC result struct; */
385          xdr_free( xdr_RE_status_result, (char *)re_ping_result);
386      }

388  }
```

```
392  /***************************************************************
393   *
394   * EDMRST_Finish
395   *
396   * Function Description:
397   *
398   * This function terminates a restoral session,   but only during the browse and
399   * mark phase.   It will be rejected if a restore is currently being executed.
400   * This routine will clean up any local memory used in the session   and will
401   * disconnect from the Restore Engine.   After calling this function,
402   * EDMRST_Initialize MUST be called before calling any other   functions in this
403   * API.
404   *
405   * Parameters:
406   *
407   * svrHdl (I) - A pointer to this user's client handle for the
408   *              Restore Engine (server) connection.
409   *
410   * Return Codes:
411   *
412   *   EP_RB_RECOVER_BAD_ARGS
413   *   EP_RB_RECOVER_RPC_FAIL
414   *   EP_RB_RECOVER_INVALOP
415   *   EP_RB_RECOVER_SERVERFAIL
         *
417   ***************************************************************/
418  eerrno_ty
419  EDMRST_Finish( serverHandle svrHdl )
420  {
421      eerrno_ty         api_status = E_SUCCESS;
422      RE_null_args      re_finish_args;
423      RE_status_result  *re_finish_result = NULL;
         int               csc_status;

425      if ( NULL == svrHdl || NULL == handlePtr
426           || svrHdl != handlePtr->re_binding_handle )
427      {
428          return( EP_RB_RECOVER_BAD_ARGS );
429      }

431      set_rpc_obj( re_finish, &re_finish_args.RPCobjID );
432      re_finish_result = re_finish_1( &re_finish_args, svrHdl );
433      if (!re_finish_result) {
434          api_status = EP_RB_RECOVER_RPC_FAIL;
435          rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL );
436      }
437      else {
438          api_status = re_finish_result->status;
439          /* release RPC result struct; */
440          xdr_free( xdr_RE_status_result, (char *)re_finish_result);
441      }

443      rec_api_log_end();   /* write last log and close the log file. */

446      return( api_status );

448  }   /* EDMRST_Finish */
```

```
%/*
%** Copyright 1997,1998 EMC Corporation
%*/

/*
**  Leading % causes rpcgen to pass a line directly thought to the output,
**  ie edmlink_sunrpc.h in this case.  This allows the .h to make a little
**  more sense and be properly documented.
*/

%/*
%** dispatch_daemon.x  : EDM Dispatch Daemon C/S communication module
%**
%** Mission Statement:  This is an RPCGEN file which defines the RPC interface
%**                     between the Dispatch Daemon (which resides on
%**                     the EDM server) and the backup client callers of its
%**                     functions.  This defines the RPC level calls that a
%**                     "caller" can make and the "service" will respond to.
%**
%** Primary Data Acted On: This defines the data that will flow over the wire.
%**                        The RPC mechanism will take care of data
%**                                                                marshalling
%**
%** Compile-Time Options:
%**   This acutally gets run through RPCGEN not compiled.  It
%**   must be run through with the -h flag to create a
%**   header, the -m flag to create the service side
%**   routines, the -l flag to create the client side
%**   routines, and the -c flag to create the common data
%**   marshalling routines.
%**
%** Basic idea here:
%**   Define the RPC level interfaces to the Dispatch Daemon
%**   and all data types that will be passed via RPC.
%*/

/**************************************************************
 * Data Structure Definitions
 **************************************************************/

/**************************************************************
 * Constant Definitions
 **************************************************************/

struct DD_rpc_objID
{
    int          type;      /* Object identifier (DD_OTYPE_ *) */
};
%#define DD_OTYPE_INIT_IN    1    /* Initialize Input Object */
%#define DD_OTYPE_INIT_OUT   2    /* Initialize Output Object */

struct DD_client_session_id {
    unsigned long   high;
    unsigned long   low;
};

const DD_SERVICE_RESTORE=1;
/* structures for input and output of re_initialize rpc call: */
struct DD_initialize_args {
    int          service;
```

```
    string hostname<>;
    string username<>;
    unsigned int timeout;
};

const DD_SERVICE_FAILURE_NONEXEC=-4;
const DD_SERVICE_FAILURE_PERMS=-2;
const DD_SERVICE_FAILURE_EXEC=-1;
const DD_SERVICE_STARTING=1;
const DD_SERVICE_RUNNING=2;
const DD_SERVICE_COMPLETED=4;

struct DD_initialize_result {
    unsigned int status;
    DD_client_session_id service_handle;
};

/* structures for getstatus function */
struct DD_getservicestatus_args {
    int                   status;
    DD_client_session_id  service_handle;
};

struct DD_getservicestatus_result {
    int     status;
    opaque  handle<128>;
};

/* work item type */
/*
 * These match the rbconfig.h for the most part.  There are
 * some extras for identifying NOS workitems.
 */
const FS_BACKUP_TYPE            = 0;
const SHARED_PART_BACKUP_TYPE   = 1;
const SHARED_M_PART_BACKUP_TYPE = 2;
const OFFLINE_DB_TYPE           = 3;
const ONLINE_KICKDB_TYPE        = 4;
const ONLINE_LISTDB_TYPE        = 5;
const DCONN_KICK_TYPE           = 6;
const DCONN_NET_TYPE            = 7;
const DCONN_WRK_TYPE            = 8;

/* length of various buffers */
const MEDNAME_SIZE =    6;
const TRLNAME_SIZE =   16;
const WINAME_SIZE =    64;
const TEMPLNAME_SIZE = 64;
const USERNAME_SIZE =  64;
const HOSTNAME_SIZE =  256;
const CLNTNAME_SIZE =  256;
const SERVER_SIZE =    256;
const MAX_STRING_SIZE = 256;  /* must be the length of the longest buffer */

/* defines for operation_type */
const BACKUP_TYPE =   1;
const RESTORE_TYPE =  2;
const OTHER_TYPE =   16;

/* work item structure */
struct WIProgress {
    unsigned long   time_started;
    unsigned long   curr_time;
    unsigned long   total_kbytes_sofar;
    unsigned long   total_files;
```

```
    unsigned long    total_badfiles;

    unsigned long    curr_kbytes_sofar;
    unsigned long    curr_time_slice;
    unsigned long    curr_files;

    unsigned long    total_files_expected;
    unsigned long    total_kb_expected;
};

struct WIProgress
{
    int              operation_type;
    int              completed;
    unsigned long    status;

    struct WIProgress    *next;

    char    wi_name[WINAME_SIZE];
    char    trail_name[TRLNAME_SIZE];
    char    trailset_name[TRLNAME_SIZE];
    char    template_name[TEMPLNAME_SIZE];
    char    client_name[CLNTNAME_SIZE];
    char    server_name[SERVER_SIZE];
    char    media_type[MEDNAME_SIZE];
    char    userid[USERNAME_SIZE];

    char    level;
    char    type;
};

/* SUMMARY structure */
struct EDMProgress {
    unsigned long    time_started;
    unsigned long    curr_time;

    unsigned long    total_kbytes_sofar;
    unsigned long    total_files;
    unsigned long    total_badfiles;

    unsigned long    curr_time_slice;
    unsigned long    curr_kbytes_sofar;
    unsigned long    curr_files;

    unsigned long    active;
    unsigned long    total;
    unsigned long    failed;
    unsigned long    successful;

    unsigned long    total_files_expected;
    unsigned long    total_kb_expected;

    int              operation_type;
    int              completed;
    unsigned long    status;

    struct EDMProgress    *next;

    char    host_name[HOSTNAME_SIZE];
};

struct EDMStats
{
    unsigned long    status;
    EDMProgress      edm;
    WIProgress       *wiprogress;
};

struct CC_Notify
```

```
{
    int       msgtype;
    int       sourcemodule;
    int       level;
    int       msglen;
    string    msgtext<>;
};

struct SessionInfo
{
    DD_client_session_id  service_handle;
    unsigned int          status;
    unsigned long         jobstarttime;
    int                   operation_type;
    long                  lastSent;
    long                  lastReceived;
    int                   outhandle;
    int                   errhandle;

    SessionInfo    *next;
};

struct SessionBlock
{
    struct SessionInfo    *sess;
    int                   totalsessions;
};

program EDM_DISPATCH_DAEMON {
    version EDMDD_FUNCTIONS {

        /* Functions for EDMRST_Initialize */
        DD_initialize_result dd_initialize( DD_initialize_args ) = 1;
        DD_getservicestatus_result dd_getservicestatus(
                        DD_getservicestatus_args ) = 2;

        SessionBlock dd_getsessioninfo( DD_getservicestatus_args ) = 3;

    } = 1; /* This is version 1 */

%/* This is the RPC program number.
%       These are reserved in /pds/docs/RPC_numbers
% * This number cannot be re-used by any other RPC daemon on the machine,    as it
% * identifies this daemon uniquely.    If it were to be re-used,    the last daemon
% * to register would be contacted when RPC's come in for this number.
%*/
%*/
= 390015;
```

```
 1  /*
 2  ** Copyright 1996, 1997 EMC Corporation
 3  */
 5  /*
 6  **
 7  ** EDMDispatch.c
 8  **
 9  ** Mission Statement: This is the main service file for the dispatch
10  **                    daemon.
11  **                    This file contains the callbacks from the main
    **                    function
    **                    which prepares the daemon to go off and service
    **                    RPC's.
12  ** Primary Data Acted On:
13  **
14  ** Compile-Time Options:
15  **
16  **                None.
17  **
18  ** Basic idea here: Module for UNIX specific daemon initialization
19  */
21  /*
22  ** The following provides an RCS id in the binary that can be located
23  ** with the what(1) utility. The intent is to keep this short.
24  */
25  #if !defined(llint)
26  static char     RCS_id [] = "@(#)$RCSfile: EDMDispatch.c,v $ "
27                              "$Revision: 1.23 $ "
28                              "$Date: 1997/02/06 20:49:15 $" ;
29  #endif
31  /* #define _POSIX_SOURCE   unable to compile with this define set */
32  /* #define _XOPEN_SOURCE   unable to compile with this define set */
34  #include <esl/c_portable.h>
35  #include <esl/ep_xopen.h>
36  #include <esl/inout.h>
38  #include <stdarg.h>
39  #include <string.h>
40  #include <syslog.h>
41  #include <pthread.h>
42  #include <thread.h>
43  #include <sys/resource.h>
45  #include <logging/logging.h>
46  #include <util/esl_core.h>
47  #include <util/esl_pidfile.h>
48  #include <EDMDispatchLog.h>
49  #include <util/esl_daemon.h>
    #include <csc/csconm.h>
51  #include <restore/csc_EDMDispatch.h>
53  #include <EDMmain.h>
54  #include <EDMDD_ccw.h>
55  #include <EDMDispatchLog.h>
56  #include <EDMDispatchBackground.h>
57  #include <EDMDDcr_rstsvc.h>
59  /*
60   * Need to define _XOPEN_SOURCE for signal funtion definitions
61   * and certain signal structure definitions.
62   */
63  #define _XOPEN_SOURCE
```

```
65  #include <signal.h>
67  #undef _XOPEN_SOURCE
69  static rpc_if_handle_t if_spec;
71  static int        G_debug = FALSE;   /* Variable which will disable forking */
73  static char      **commandlineargs; /* Pointer to command line args */
75  /*******************************************************************************
76  **
77  ** Routine: IsDebugOn
78  **
79  ** Inputs: None
80  **
81  ** Outputs: None
82  **
83  ** Return Codes:
84  **              TRUE if debug is on.
85  **
86  ** Purpose:     This routine can be used to tell other subsystems
87  **              whether debugging is available.
88  **
89  ** Intended caller: internal only.
90  **
91  *******************************************************************************
92  */
93  boolean_ty
94  IsDebugOn()
95  {
96      return G_debug;
97  }
```

```
 99     /**********************************************************************
100     **
101     ** Routine: kill_handler
102     **
103     ** Inputs: int sigval - the signal which was received.
104     **
105     ** Outputs: Will log messages telling what action is being taken.
106     **
107     ** Return Codes:
108     **         exits with the number of the signal received
109     **
110     ** Purpose:
111     **         This routine handles specific signals i.e. SIGINT,
112     **         SIGQUIT,
113     **         SIGTERM. Each results in a log entry and an exit.
114     **
115     ** Intended caller: internal only.
116     **
117     **         *********
118     */
119     static void kill_handler( IN int sigval )
120  1  {
121  1      int      status;
122  1      time_t   current_time;
            char     *ctimebuf;
            char     *ebuff;

124  1      /* If main exits, it calls this routine with signal 0 */

126  1      /* Unregister the interface */
127  1      (void) csc_unregister_server_interface(&if_spec, &status);

129  1      /* If the unregister fails, report the problem, but continue */
130  2      if ( status != error_status_ok )
131  2      {
132  2          ebuff = (char *) csc_get_error( status );

134  2          (void) EDMDispatch_logent(
135  2              __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
136  2              "CSC_SERVER_LOGIN failed: <%d> %s",
137  1              status, (ebuff ? ebuff : "Unknown error") );

139  1      }

140  1      /* Get the current time */
142  1      (void) time(&current_time);

144  1      ctimebuf = ctime(&current_time);

145  1      /* Overlay newline with null - buf should always be 26 bytes long */
147  1      ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

148  1      (void) EDMDispatch_logent(
150  1          __FILE__, __LINE__, LOG_INFO, MESSAGE_SHUTDOWN, 0,
151  1          "Shutting down at %s due to signal %d", ctimebuf,
152  1          sigval);
153  1      /*
155  1       * Remove our lock file.
157  1       */
            (void) EslDestroyPidFile(PIDPATH);

            exit(sigval);

        } /* End of kill_handler() */
```

```
159     /**********************************************************************
160     *
161     * Function Name:
162     *         display_usage
163     *
164     * Simply displays the usage
165     *
166     * Call Arguments:
167     *         Program name
168     *
169     * Error Outputs and Side Effects:
170     *         Prints usage.
171     *
172     * Special Considerations:
173     *         None.
174     *
175     *         *********
176     */
177     static void
178     display_usage (IN char *progname)
179  1  {
181  1      /* Print out usage stmt. */
182  1      fprintf (stderr, "Usage: %s [-d]\n", progname);
184  1      fprintf (
            stderr, "-d keep the daemon from forking so debugging is easier\n");

        } /* end display_usage () */
```

```c
187  /*************************************************************
188  **
189  ** Routine:  daemon_catch_interrupts
190  **
191  ** Inputs:
192  **
193  **          None
194  **
195  ** Return Codes:
196  **
197  **          None
198  ** Purpose:  Sets up signals for service. On NT we will have to
199  **           consider what OS constructs to replace signals with.
200  **           In this case we are catching SIGTERM, SIGINT, and
201  **           SIGQUIT and ignoring anything else.
202  **
203  ** Outputs:
204  **
205  **          None
206  **
207  ** Intended caller:  internal only.
208  ************************************************************
     */
209  void daemon_catch_interrupts()
     {
211      struct sigaction     sactions;     /* Signal actions */

213      ZERO( sactions );

214      /*
215       * Set an empty list so we can set signals we want to handle
216       */
218      (void) sigemptyset( &sactions.sa_mask );

219      /*
220       * Add signals that we want to handle
221       */
222      (void) sigaddset( &sactions.sa_mask, SIGTERM );
223      (void) sigaddset( &sactions.sa_mask, SIGINT );
225      (void) sigaddset( &sactions.sa_mask, SIGQUIT );

226      /* Setup the signal handler. */
228      sactions.sa_handler = kill_handler;

229      /*
230       * Assign handler to each signal we are interested in.
231       */
232      (void) sigaction( SIGTERM, &sactions, NULL );
233      (void) sigaction( SIGINT,  &sactions, NULL );
235      (void) sigaction( SIGQUIT, &sactions, NULL );

236      /*
237       * Setup mask so we can specify what signals we will ignore.
238       */
240      (void) sigfillset( &sactions.sa_mask );

241      /*
242       * We want to ignore everything except those we have set up
243       * above so to remove those from the list.
244       */
245      (void) sigdelset( &sactions.sa_mask, SIGTERM );
246      (void) sigdelset( &sactions.sa_mask, SIGINT );
248      (void) sigdelset( &sactions.sa_mask, SIGQUIT );

         /*
```

```c
249      * Set the mask. Since no other threads have been started,
250      * all threads will get this mask.
251      */
252      (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
253  }
```

```
256
257   /************************************************************
258   **
259   ** Routine: daemon_check_proper_ID
260   **
261   ** Inputs:        None
262   **
263   ** Outputs:       None
264   **
265   ** Return Codes:
266   **            exits with an error when the user is not root
267   **
268   ** Purpose:
269   **            Checks user's ID and determines if the user is allowed
270   **            to execute service.
271   **            If there are no constraints then this
272   **            function may be blank.
273   **
274   ** Intended caller: internal only.
275   **
276   *************************************************************
277   */
278   void daemon_check_proper_ID()
279   {
280       /*
281       ** Check for root
282       */
283 1     if (geteuid() != E_ROOTUID)
284 2     {
285 2         (void) EDMDispatch_logent(
286 2             __FILE__, __LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
287 2             "Must be run as superuser, uid was %d",
288 1             geteuid());
289 1         exit(1);
           }
       }
```

```
291
292   /************************************************************
293   **
294   ** Routine: parse_commandline
295   **
296   ** Inputs:      argc, argv (command line arguments)
297   **
298   ** Outputs:     None
299   **
300   ** Return Codes:
301   **           exits with an error when the user types a bad argument
302   **
303   ** Purpose:
304   **           Parses command line arguments and sets flags. If there
305   **           are no flags to be set then this function may be empty.
306   **
307   ** Intended caller: internal only.
308   **
309   *************************************************************
310   */
311 1 void parse_commandline(int argc, char *argv[])
312 1 {
313 1 int    opt;                    /* Process options   */
314 1
316 1 commandlineargs = argv;
317 2
318 2 while ((opt = getopt(argc,argv,"dD")) != EOF )
319 3 {
320 3     switch(opt)
321 3     {
322 3         case 'd':
323 3         case 'D':
324 3             G_debug = TRUE;
325 3             break;
326 3         default:
            (void) display_usage( argv[0] );
            exit(1);
328 2     }
329 1 }
330   }
```

```
332  /***********************************************************************
333  **
334  ** Routine: daemon_initialize_logging
335  **
336  ** Inputs:      None
337  **
338  ** Outputs:     None
339  **
340  ** Return Codes:
341  **              None
342  **
343  ** Purpose:     Do whatever it takes to initialize logging.  In the near
344  **              future this may involve doing something with catalogs or
345  **              calling higher level logging functions which encapsulate
346  **              these things.
347  **
348  ** Intended caller:  internal only.
349  **
350  **
351  ***********************************************************************/

353  void
354  daemon_initialize_logging()
355  {
356      /* Pass in argv[0], the program name */
357      (void) esl_log_init(commandlineargs[0]);
358  }
```

```
360  /***********************************************************************
361  **
362  ** Routine: daemon_become_daemon
363  **
364  ** Inputs:      None
365  **
366  ** Outputs:     None
367  **
368  ** Return Codes:
369  **              exits with an error code if initialization fails
370  **
371  ** Purpose:     This function is for doing the forking etc. under UNIX.
372  **              It is unknown what will be necessary under NT.
373  **
374  ** Intended caller:  internal only.
375  **
376  **
377  ***********************************************************************/

379  void
380  daemon_become_daemon()
381  {
382      char *ptr;
383      int ret = 0;

385      /*
386       * Strip the path from the program name so we can use it
387       * elsewhere.
388       */
389      ptr = strrchr(commandlineargs[0], '/');
390      if (ptr == NULL)
391          ptr = commandlineargs[0];
392      else
393          ptr++;

395      /* Change directory to a process specific core directory */
396      ret = esl_coredir_setup(ptr);
397      if (ret != 0)
398      {
399          (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
400                  MESSAGE_ERR_IN_ESL_COREDIR, 0,
401                  "esl_coredir_setup failed: <%d>",
402                  errno);
403          exit(1);
404      }

406      /*
407       ** This is now esl functionality.
408       ** This code does everything necessary
409       ** to make this a "real" daemon by detaching from the
           terminal
410       ** changing the process group, closing stdout, stderr, stdin,
           ...
           */
412      if (G_debug == FALSE)
413      {
414          ret = esl_daemon_startup();
415          if (ret != 0)
416          {
417              fprintf(
418                  stderr, "%s: Failed to initialize as daemon.\n",
                         commandlineargs[0]);
```

```
419  3
420  2              exit(1);
421  1          }
422            }
```

---

```
424  1  /***********************************************************************
425  1  **                                                                   ***
426  1  ** Routine: rpc_init
427  1  **
428  1  ** Inputs:    None
429  1  **
430  1  ** Outputs:   None
431  1  **
432  1  ** Return Codes:
433  1  **            exits with an error code if initialization fails
434  1  **
435  1  ** Purpose:   This function is for doing RPC initialization.
436  1  **            For the most part it involves calling the csc routines.
437  1  **            This is pretty standard between UNIX and NT.
438  1  **
439  1  ** Intended caller:  internal only.
440  1  **
441  1  ***********************************************************************
442  1  */

444  1  void rpc_init()
445  1  {
446  1  error_status_t         status;          /* error status (nbase.h) */
447  1  char     *ebuff;

449  1  /*
450  1  ** This is here because of HP which may or may not define timeval.
451  1  ** May be removed when esl_timeval is ported to clients
452  1  */
453  1  #ifdef _STRUCT_TIMEVAL
454  1  struct timeval     sleep_interval = {5,0};   /* 5 second sleep interval */
455  1  #else
456  1  struct timespec sleep_interval = {5,0};   /* 5 second sleep interval */
457  1  #endif

459  1  /* Setup the interface specification for RPC */
460  1  SERVER_IFSPEC(if_spec);

462  1  /*
463  1  * Login as SERVER_PRINCIPAL.  The context of the process
464  1  * will be set to this principal.
465  1  *
466  1  * This process will keep trying to login to DCE if the
467  1  * server is unavailable.
468  1  *
469  1  * Note that under SUN RPC this is a no-op.
470  2  */
471  2  while (TRUE)
472  2  {
474  2      (void) csc_server_login(SERVER_PRINCIPAL,
                                   SERVER_KEYTAB, &status);
475  2      /* If we succeeded, then exit this loop. */
476  3      if ( status == error_status_ok )
477  3      {
478  3          break;
479  2      }
480  3      else     /* Print error message if appropriate. */
481  3      {
            ebuff = (char *) csc_get_error( status );
```

```
483  3                  (void) EDMDispatch_logent(
                                __FILE__, __LINE__, LOG_ERR,
                                MESSAGE_NO_LOGIN, 0,
                                "CSC_SERVER_LOGIN failed: <%d>
                                %s",
484  3                          status, (
485  3                          ebuff ? ebuff : "Unknown error"));
486  3          }
487  2

489  2          /* If the failure was due to unavailable client,
490  2           * pause and then try again.
491  2           */
492  2          if (status == sec_rgy_server_unavailable)
493  3          {
494  3                  /*
495  3                   * uses sleep when SUNRPC, otherwise uses
496  3                   * pthread call to delay for the specified
497  3                   * time
498  3                   */
499  3                  CSC_SLEEP(sleep_interval);
500  3                  continue;
501  2          }

503  2          /* If we got here, we had a unexpected failure.  */
504  2          (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
505  2                          MESSAGE_NO_LOGIN, 0,
506  2                          "The service cannot log in as
                                required");

508  2          }
509  1

511  1          /*
512  1          ** We need to initialize the authorization module before we
                   do
513  1          ** a listen.
514  1          */
515  1          (void)csc_authorization_init(&status);

517  1          if ( status != error_status_ok )
518  2          {
519  2                  ebuff = (char *) csc_get_error( status );

521  2                  (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
522  2                          MESSAGE_NOAUTHORIZATION, 0,
523  2                          "CSC_AUTHORIZATION_INIT failed: <%d> %s",
524  2                          status, (
525  2                          ebuff ? ebuff : "Unknown error") );
526  1          }

                exit(1);

529  1          (void) csc_register_server_interface(
530  1                          &if_spec,
531  1                          SERVER_ANNOTATION,
                                &status);

533  1          if ( status != error_status_ok )
534  2          {
535  2                  ebuff = (char *) csc_get_error( status );

537  2                  (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
538  2                          MESSAGE_CANNOTREGISTER, 0,
539  2                          "CSC_REGISTER_SERVER_INTERFACE failed:
                                <%d> %s",
540  2                          status, (
```

```
                        ebuff ? ebuff : "Unknown error") );

541  2                  ebuff ? ebuff : "Unknown error") );
542  1          }
543             exit(1);
        }
```

```
/*********************************************************************
**
** Routine: rpc_run
**
** Inputs:
**     None
**
** Outputs:
**     None
**
** Return Codes:
**     None
**
** Purpose:
**     This function is for running the RPC listen.
**     This is pretty standard between UNIX and NT.
**
** Intended caller: internal only.
**
*********************************************************************/

void rpc_run()
{
    error_status_t  status;         /* error status (nbase.h) */
    char            *ebuff;

    /* listen for RPC calls forever. */
    csc_server_listen(
        rpc_c_listen_max_calls_default, &status );

    /* We don't expect to get here. */
    (void) EDMDispatch_logent(
        __FILE__, __LINE__, LOG_ERR, MESSAGE_SERVERLISTEN, 0,
        "CSC_SERVER_LISTEN failed: <%d> %s",
        status, (
        ebuff = (char *) csc_get_error( status );
        ebuff ? ebuff : "Unknown error") );
}
```

```
/*********************************************************************
**
** Routine: daemon_specific_initialization
**
** Inputs:
**     None
**
** Outputs:
**     None
**
** Return Codes:
**     None
**
** Purpose:
**     Do whatever makes this daemon special.  In some cases you
**     may want to start a thread or open a socket.  Do that here.
**
** Intended caller: internal only.
**
*********************************************************************/

void
daemon_specific_initialization()
{
    error_status_t  status;         /* error status (nbase.h) */

    int             ret;
    pthread_t       mantid;
    pthread_t       cleantid;
    time_t          current_time;
    char            *ctimebuf;
    struct rlimit   rlp;

    /* Create a file and lock it so we don't start multiple
    ** daemons.  Exit if there is another copy of us running.
    ** The CreatePidFile call already logs errors so just exit.
    */
    if (EslCreatePidFile(PIDPATH))
    {
        exit(1);
    }

    /* Find out what time it is */
    (void) time(&current_time);

    ctimebuf = ctime(&current_time);

    /* Overlay newline with null - buf should always be 26 bytes
       long */
    ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

    /* Log startup message */
    (void) EDMDispatch_logent(
        __FILE__, __LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
        "Restore service %s starting up at %s",
        commandlineargs[0],
        ctimebuf );

    /* set the open files limit very large */
    rlp.rlim_max = FD_SETSIZE;
    rlp.rlim_cur = FD_SETSIZE;
    setrlimit(RLIMIT_NOFILE, &rlp);
```

```
638  1      getrlimit(RLIMIT_NOFILE, &rlp);

640  1      (void) EDMDispatch_logent(
                __FILE__, __LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
                "Service allows %d open files",
                rlp.rlim_max );
641  1

643  1      /* Initialize service launcher */
644  1      ret = EDMDDSvcInit();

646  1      if (ret != 0)
647  2      {
648  2          (void) EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_INFO, 0, 0,
                    "Service Launcher failed returning - %d",
                    ret);
649  2
650  2          exit(1);
651  1      }

653  1      /*
654  1       * Start the other threads in the daemon. The main thread
655  1       * becomes the RPC thread. BAMDataManage is the entry point
656  1       * for the data collection thread. BAMDataCleanup is the
657  1       * entry point for the data expiration thread.
658  1       */
659  1      /*pthread_create(&mantid, NULL, DispDaemon_ccr, NULL); */
660  1      pthread_create(&mantid, NULL, DispDaemon_ccw, NULL);
661  1      pthread_create(&cleantid, NULL, DispatchBackground, NULL);
662  1      rpc_init();
663  1      rpc_run();
664  1      }
```

```
666      /*********************************************************************
667      **
668      ** Routine: daemon_cleanup
669      **
670      ** Inputs:      None
671      **
672      ** Outputs:     None
673      **
674      ** Return Codes:
675      **              None
676      **
677      ** Purpose:      Call function which will clean up daemon properly.
678      **
679      ** Intended caller:  internal only.
680      **
681      ** ******************************************************************
682      */
684      void
685      daemon_cleanup()
686      {
687  1       kill_handler( 0 );
688  1      }
```

```
 1  /*
 2  ** Copyright 1996,1997 EMC Corporation
 3  */

 6  /*
 7  ** EDMDispatchService.c
 8  *
 9  * Mission Statement:   RPC entry points.
10  *
11  * Primary Data Acted On:
12  *
13  * Compile-Time Options:
14  *
15  * Basic idea here:
16  */

18  #if !defined(lint)
19  static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
20                              "$Revision: 1.0 $ "
21                              "$Date: 1997/02/06 20:49:15 $" ;
22  #endif

24  #include <eslc_portable.h>
25  #include <esl/inout.h>

27  #include <logging/logging.h>
28  #include <csc/csconn.h>

30  #include <restore/csc_EDMDispatch.h>
31  #include <restore/dispatch_daemon.h>

33  #include <EDMDispatchLog.h>
34  #include <EDMDispatchSession.h>

36  /*
37  * These are all the rpc entry points for the dispatch daemon.
38  * The dispatch daemon is multi-threaded and it is the main thread
39  * which handles all incoming RPC. ONC RPC is single threaded
40  * so each call blocks other RPC calls. This provides us some
41  * safety in the way we handle our data and limits our exposure
42  * to unexpected multithreading problems.
43  *
44  */

46  static void FreeSessionInfo(SessionInfo *);

47  /********************************************************
48  **
49  ** Routine:  dd_initialize_1
50  **
51  ** Inputs:   DD_initialize_args * - args for the restore initialize
                                        call
52  **
53  ** Outputs:  None
54  **
55  ** Return Codes:
56  **           DD_initialize_result * - result of init function call
57  **
58  ** Purpose:  Function to create a restore session.
59  **
60  ** Intended caller:  Internal Only.
61  ********************************************************/
63  DD_initialize_result *
```

```
64    dd_initialize_1_svc(
                            IN DD_initialize_args *arg, IN struct svc_req *req )
65  1 {
66  1     static DD_initialize_result argzz;

68  1     InitializeSession(arg, req, &argzz);

70  1     return &argzz;
71  1 }
```

```
73   /**********************************************************************
74   **
75   ** Routine:  dd_getservicestatus_1
76   **
77   ** Inputs:   DD_getservicestatus_args * - args for the
                   getservicestatus call
78   **
79   ** Outputs:  None
80   **
81   ** Return Codes:
82   ** DD_getservicestatus_result * - result of status function
                   call
83   **
84   ** Purpose:  Function to poll for status on a session.
85   **
86   ** Intended caller:  Internal Only.
87   **********************************************************************
88   */

90   DD_getservicestatus_result *
91   dd_getservicestatus_1_svc(
91       IN DD_getservicestatus_args *arg, IN struct svc_req *req )
92   {
93 1     static DD_getservicestatus_result argzz;

95 1     GetDispatchStatus(arg, &argzz);

97 1     return &argzz;
98   }
```

```
100  /**********************************************************************
101  **
102  ** Routine:  dd_getsessioninfo_1
103  **
104  ** Inputs:   DD_getservicestatus_args * - args for the getsessioninfo
                   call
105  **
106  ** Outputs:  None
107  **
108  ** Return Codes:
109  ** SessionBlock * - result of session info call
110  **
111  ** Purpose:  Function to get information on all sessions.
112  **
113  ** Intended caller:  Internal Only.
114  **********************************************************************
115  */

117  SessionBlock *
118  dd_getservicestatus_1_svc(
119 1    IN DD_getservicestatus_args *arg, IN struct svc_req *req )
120 1 {
121 1    static SessionBlock argzz;
         static boolean_ty first = TRUE;

123 1    if (first)
124 2    {
125 2       memset(&argzz, 0, sizeof(argzz));
126 2       first = FALSE;
127 1    }
128 1    else
129 2    {
130 2       FreeSessionInfo(argzz.sess);
131 2       argzz.sess = NULL;
132 1    }

134 1    GetDispatchInfo(arg, &argzz);

136 1    return &argzz;
137  }
```

```
139
140   /****************************************************************
141   **
142   ** Routine:  FreeSessionInfo
143   **
144   ** Inputs:   SessionInfo * - arg to free
145   **
146   ** Outputs:  None
147   **
148   ** Return Codes:
149   **           None
150   **
151   ** Purpose:  Function to free all SessionInfo structures in a list.
152   **
153   ** Intended caller: Internal Only.
      ****************************************************************/
155   static void FreeSessionInfo(SessionInfo *sess)
156 1 {
157 1    if (sess == NULL)
158 1       return;

160 1    if (sess -> next != NULL)
161 1       FreeSessionInfo(sess -> next);

163 1    free(sess);
164 1 }
```

```
1   /*
2   ** Copyright 1996,1999 EMC Corporation
3   */

5   /*
6   **
7   ** EDMDispatchSession.cc
8   **
9   ** Mission Statement: This is where all session management occurs.
10  **
11  ** Primary Data Acted On:
12  **
13  ** Compile-Time Options:
14  **
15  **          USE_SUNRPC - Compile source with sunrpc
16  **                       support.  If
17  **                       not set, assume DCE support.
18  **
    ** Basic idea here: Module for session management
20  */

21  /*
22  ** The following provides an RCS id in the binary that can be located
23  ** with the what(1) utility. The intent is to keep this short.
24  */
25  #if !defined(llint)
26  static char   RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ "
27                            "$Revision: 1.23 $ "
28                            "$Date: 1997/02/06 20:49:15 $ " ;
    #endif

30  /* #define _POSIX_SOURCE    unable to compile with this define set */
31  /* #define _XOPEN_SOURCE    unable to compile with this define set */

33  #include <esl/c_portable.h>
34  #include <esl/ep_xopen.h>
35  #include <esl/inout.h>

37  #include <pthread.h>
38  #include <memory.h>
39  #include <sys/time.h>
40  #include <sys/types.h>
41  #include <syslog.h>

43  // Rogue Wave includes
44  #include <rw/collect.h>
45  #include <rw/rwfile.h>
46  #include <rw/vstream.h>
47  #include <rw/bintree.h>

49  #include <csc/cscomm.h>
50  #include <restore/dispatch_daemon.h>
51  #include <restore/dispatch_protocol_client.h>
52  #include <EDMSession.h>
53  #include <EDMReturnMessageApi.h>
54  #include <EDMDHandleMgrApi.h>
55  #include <EDMDispatchSession.h>
56  #include <EDMDispatchConfig.h>
57  #include <EDMDDcr_rstsvc.h>

59  #include <EDMDispatchLog.h>

61  static RWBinaryTree       G_sessionTree;

63  static pthread_mutex_t    G_sessionTreeMtx = PTHREAD_MUTEX_INITIALIZER;
64  extern ElinkHandlePtr_ty  ElinkHandle;
```

```
66  static int maxDisconnectTime = SECONDS_PER_HOUR;  // one hour

68  /*****************************************************************
69  **
70  ** Routine:     LockSessionMutex
71  **
72  ** Inputs:      None
73  **
74  ** Outputs:     None
75  **
76  ** Return Codes:
77  **              None
78  **
79  ** Purpose:     Lock the session mutex.
80  **
81  *****************************************************************
82  */

84  static void
85  LockSessionMutex()
86  {
87      static boolean_ty first = TRUE;

89      if (first == TRUE)
90      {
91          first = FALSE;
92          pthread_mutex_init(&G_sessionTreeMtx, NULL);
93      }

95      pthread_mutex_lock(&G_sessionTreeMtx);
96  }
```

```
 98   /*****************************************************************************
 99   **
100   ** Routine:   UnlockSessionMutex
101   **
102   ** Inputs:    None
103   **
104   ** Outputs:   None
105   **
106   ** Return Codes:
107   **    None
108   **
109   ** Purpose:   Unlock the mutex for the session tree object
110   **
111   *****************************************************************************
112   */
114   static void
115   UnlockSessionMutex()
116 1 {
117 1   pthread_mutex_unlock(&G_sessionTreeMtx);
118 1 }
```

```
120   /*****************************************************************************
121   **
122   ** Routine:   InitializeSession
123   **
124   ** Inputs:    DD_initialize_args *arg - args sent via RPC for starting
125   **                                      session
126   **            struct svc_req *req - the request block from RPC
127   **
128   ** Outputs:   DD_initialize_result *res - the result structure which
129   **                                        tells whether
130   **              operation succeeded or failed.
131   **
132   ** Return Codes:
133   **    None
134   **
135   ** Purpose:   Initialize a session for the GUI.
136   **
137   *****************************************************************************
138   */
139   void
140   InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
141 1     OUT DD_initialize_result *res)
142 1 {
143 1   EDMSession  *session;
144 1   EDMSession  *ret;
145 1   pthread_t   id;
146 1   time_t      t;
147 1
148 1   if (arg == NULL || req == NULL || res == NULL)
149 2   {
150 2     return;
151 1   }
152 1
154 1   t = time(NULL);
156 1
154 1   session = new EDMSession();
156 1
156 1   if (session == NULL)
157 2   {
158 2     res -> status = DD_SERVICE_FAILURE_NONEXEC;
159 2     return;
160 1   }
162 1
162 1   session -> initSession();
164 1
164 1   session -> setStartTime(t);
166 1
166 1   session -> setOperationType(arg -> service);
168 1
168 1   session -> setStatus(DD_SERVICE_STARTING);
170 1
170 1   if (arg -> username != NULL && arg -> hostname != NULL)
171 2   {
172 2     switch(arg -> service)
173 3     {
174 3       // code is commented out because we do not
175 3       // want to read tthe config for permission information
176 3       // at this time, it is a waste of cycles
177 3 #if 0
178 3       case DD_SERVICE_RESTORE : boolean_ty allowed;
180 3         allowed =
```

```
181  3
183  3     #endif
184  4
185  4             default: // Add some error message for unknown service
186  4                 break;
187  4
188  3         }
190  3         break;
191  3
192  3     DispatchCheckRestorePermission(
193  3         arg->hostname,
194  2         arg -> username);
195  1
196  1     if (!allowed)
197  2     {
198  2         res -> status =
199  2             DD_SERVICE_FAILURE_PERMS;
200  2         delete session;
201  1         return;
           }
203  1     else
           {
205  1         res -> status = DD_SERVICE_FAILURE_NONEXEC;
               delete session;
               return;
207  1     }

           ret = (EDMSession *) G_sessionTree.insert((
               RWCollectable *) session);
209  1
210  2     LockSessionMutex();
211  2
212  2     UnlockSessionMutex();
213  2
214  1     if (ret == NULL)
216  1     {
218  1         session -> getSessionID(&res -> service_handle);
219  1     }
221  1     // Call Steve's thread
           pthread_create(&sid, NULL, &DDRStsvc_init, (void *) session);
223  1     session -> setThreadID(id);
224  1     return;
           }
```

```
226  /*****************************************************************************
227  **
228  ** Routine:   SendPingMessagesToSession
229  **
230  ** Inputs:    None
231  **
232  ** Outputs:   None
233  **
234  ** Return Codes:
235  **            None
236  **
237  ** Purpose:   Queue up all the ping messages to the sessions.  If they don't
238  **            respond they should be considered dead.
239  **
240  **
241  *****************************************************************************/
243  void
     SendPingMessagesToSession()
244  {
245      EDMSession *sess;
246
248      LockSessionMutex();
250      RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                 G_sessionTree);
252      while ( sessionIterator != NULL &&
253          (sess = (EDMSession*) (*sessionIterator)()) != NULL )
254      {
255          DD_client_session_id sid;
256          rpc_binding_handle_t *cscb = NULL;
257          int                  status;
258          int                  ret;
260          if (sess -> getStatus() != DD_SERVICE_RUNNING)
261              continue;
263          sess -> getSessionID(&sid);
265          ret = GetCSCHandle(&sid, &cscb, &status);
267          if (ret != 0 || cscb == NULL || *cscb == NULL)
268              continue;
270          PushResponseMessage(dp_ping_request, sid, cscb, &status);
271      }
273      // through with iterator
274      if (sessionIterator != NULL)
275      {
276          delete sessionIterator;
277      }
279      UnlockSessionMutex();
280  }
```

```
282  /****************************************************************
283  **
284  ** Routine:  UpdateSessionLastReceived
285  **
286  ** Inputs:   DD_client_session_id *sessID - session that sent us
287  **                                          something
288  **
289  ** Outputs:  None
290  **
291  ** Return Codes:
292  **     0 on success and non-zero otherwise
293  **
294  ** Purpose:  Update the specified session with the lastest received
295  **           message
296  **           time.
297  **
299  ****************************************************************/
300  */
301  int
     UpdateSessionLastReceived(DD_client_session_id *sessID)
301  {
302      time_t    last = time(NULL);
303      EDMSession    *session;
304      EDMSession    *ret;

306      session = new EDMSession();

308      if (session == NULL)
309      {
310          EDMDispatch_logent(
311              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
312              "Failure to create a session block");
313          return -1;
315      }

317      session -> setSessionID(sessID);

319      LockSessionMutex();

321      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

323      UnlockSessionMutex();

323      delete session;

325      if (ret == NULL)
326      {
327          EDMDispatch_logent(
328              __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
329              "Failure to update session %ld:%ld received
330              time",
331              sessID -> high, sessID -> low);
333          return -1;
335      }

         ret -> setLastReceived(last);

336      return 0;
     }
```

```
338  /****************************************************************
339  **
340  ** Routine:  UpdateSessionLastSent
341  **
342  ** Inputs:   DD_client_session_id *sessID - session that sent us
343  **                                          something
344  **
345  ** Outputs:  None
346  **
347  ** Return Codes:
348  **     0 on success and non-zero otherwise
349  **
350  ** Purpose:  Update the specified session with the lastest sent
351  **           message
352  **           time.
353  **
355  ****************************************************************/
356  */
357  int
     UpdateSessionLastSent(DD_client_session_id *sessID)
357  {
358      time_t    last = time(NULL);
359      EDMSession    *session;
360      EDMSession    *ret;

362      session = new EDMSession();

364      if (session == NULL)
365      {
366          EDMDispatch_logent(
367              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
368              "Failure to create a session block");
369          return -1;
371      }

373      session -> setSessionID(sessID);

375      LockSessionMutex();

377      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

379      UnlockSessionMutex();

381      delete session;

382      if (ret == NULL)
383      {
384          EDMDispatch_logent(
385              __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
386              "Failure to update session %ld:%ld sent
387              time",
389              sessID -> high, sessID -> low);
391          return -1;
         }

         ret -> setLastSent(last);

392      return 0;
     }
```

```
394  1   /**********************************************************************
396  3   **
397  3   ** Routine:  CheckDispatchSessions
398  3   **
399  3   ** Inputs:   None
400  3   **
401  3   ** Outputs:  None
402  3   **
403  3   ** Return Codes:
404  3   **           None
405  3   **
406  3   ** Purpose:  Look for dead sessions and kill them off
407  3   **
408  3   */
         ********************************************************************/
410  4   void
411  4   CheckDispatchSessions()
412  1   {
413  1
414  1       EDMSession      *sess;
415  1       int             status = 0;
416  1       int             ret = 0;
417  1       time_t          currTime;
         1       RWBinaryTree    reaperTree;
419  1       currTime = time(NULL);
421  1       LockSessionMutex();
423  1       RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                       G_sessionTree);
425  1       while ( sessionIterator != NULL &&
426  2           (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
428  2           if ( (sess->getLastReceived()
                         <= currTime - maxDisconnectTime() != 0) ||
429  2                (sess->getStartTime() <= currTime - maxDisconnectTime &&
430  2                ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus(
431  2                ) == DD_SERVICE_FAILURE_EXEC ||
432  3                sess -> getStatus() == DD_SERVICE_FAILURE_PERMS) )
433  3           {
434  3               // Insert it into the reaper tree
435  3               (void) reaperTree.insert(sess);
             3           }
436  1       }
438  1       // through with iterator
439  1       if (sessionIterator != NULL)
440  2       {
441  2           delete sessionIterator;
442  1       }
444  1       UnlockSessionMutex();
446  1       // If the reaper tree has something in it then use those entries
447  1       //        to remove
448  1       // things from the query tree.
449  2       if (reaperTree.entries() > 0)
450  2       {
                 sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```
452  2       while ( sessionIterator != NULL &&
453  3           (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
454  3           DD_client_session_id  sessID;
456  3           sess -> getSessionID(&sessID);
458  3           ret = removeSession(&sessID, &status);
460  3           if (ret != 0)
461  4           {
462  4               EDMDispatch_logent(
463  4                   __FILE__, __LINE__, LOG_ERR, 0, 0,
464  4                   "Failure to remove session %ld:%ld",
465  4                   sessID.high, sessID.low);
466  4               continue;
467  3           }
468  4           else
469  4           {
470  4               EDMDispatch_logent(
471  4                   __FILE__, __LINE__, LOG_INFO, 0, 0,
472  3                   "Removing session %ld:%ld. Current %ld",
473  3                   sessID.high, sessID.low,
                         sess -> getLastReceived(),
                         currTime - maxDisconnectTime);
475  3               ret = deleteHandleSet(&sessID, &ELinkHandle, &status);
477  3               if (ret != 0)
478  4               {
479  4                   EDMDispatch_logent(
480  4                       __FILE__, __LINE__, LOG_ERR, 0, 0,
481  4                       "Failure to delete handles for
                                 session %ld:%ld",
                             sessID.high, sessID.low);
483  2               }
485  2           }
486  2       }
487  3       // through with iterator
488  3       if (sessionIterator != NULL)
489  2       {
491  2           delete sessionIterator;
492  1       }
493  1       reaperTree.clear();
         }
```

```
495      /****************************************************************
496      **
497      ** Routine:  DrainSessionDescriptors
498      **
499      ** Inputs:   None
500      **
501      ** Outputs:  None
502      **
503      ** Return Codes:
504      **     None
505      **
506      ** Purpose: Drain whatever data is on stdout and stderr for sessions.
507      **
508      **
509      ****************************************************************/
511    1 void
512    1 DrainSessionDescriptors()
513    1 {
514    1     int      hout = 0, herr = 0, status = 0;
515    1     int      selret = 0;
516    1     int      i = 0;
517    1     char     buff[1024];
518    2     struct timeval timetowait = {
519    1         1, 0
520    1     };
521    1     fd_set   stdoutSet;
522    1     fd_set   stderrSet;

524    1     getStdoutSet(&stdoutSet, &hout, &status);

526    1     if ( (selret = select(
527    2           hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
528    2     {
529    3         for (; i < hout+1; i++)
530    3         {
531    4             if (FD_ISSET(i, &stdoutSet))
532    4             {
533    3                 while (read(i, buff, 1024) > 0);
534    2             }
535    1         }
           }

537    1     getStderrSet(&stderrSet, &herr, &status);

539    1     if ( (selret = select(
540    2           herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
541    2     {
542    3         for (i = 0; i < herr+1; i++)
543    3         {
544    4             if (FD_ISSET(i, &stderrSet))
545    4             {
546    3                 while (read(i, buff, 1024) > 0);
547    2             }
548    1         }
549        }
```

```
551      /****************************************************************
552      **
553      ** Routine:  GetSessionStatus
554      **
555      ** Inputs:   DD_client_session_id *ssid - session ID to check the
                      status of
556      **
557      ** Outputs:  int *status - status of the function call
558      **           int *s_status - session status
559      **
560      ** Return Codes:
561      **     0 if successful and non-zero otherwise
562      **
563      ** Purpose: Get status on the session.
564      **
565      ****************************************************************/
566    1 int
567      GetSessionStatus(
568    1     DD_client_session_id *ssid, int *s_status, int *status)
569    1 {
570    1     EDMSession   *sess;
571    1     EDMSession   *ret;

573    1     if (status == NULL)
574    2     {
575    2         return -1;
576    1     }

578    1     if (ssid == NULL || s_status == NULL)
579    2     {
580    2         *status = SESSION_BAD_ARGS;
582    2         return -1;
583    1     }

585    1     sess = new EDMSession();

587    1     if (sess == NULL)
588    2     {
589    2         EDMDispatch_logent(
                     __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                     "Failure to create a session block");
590    2         *status = SESSION_NO_MEMORY;
591    2         return -1;
593    2     }
594    1

596    1     sess -> setSessionID(ssid);

598    1     LockSessionMutex();

600    1     ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

602    1     UnlockSessionMutex();

604    1     delete sess;

606    1     if (ret == NULL)
607    2     {
608    2         EDMDispatch_logent(
                     __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                     "Failure to lookup session %ld:%ld",
```

```
610  2          *status = SESSION_LOOKUP_FAILED;
                    ssid -> high, ssid -> low);
611  2          }
612  1          return -1;
613  1      }

615  1      *s_status = ret -> getStatus();

617  1      return 0;
618  1  }
```

```
620  /*************************************************************
     **                                                        *
     **                                                        *
     ** Routine:    GetDispatchStatus                          *
     **                                                        *
     ** Inputs:     DD_getservicestatus_args *arg - session ID to check the
     **                                             status of
     **                                                        *
     ** Outputs:    DD_getservicestatus_result *res - the result structure
     **                                               which tells
     **                              whether operation succeeded or failed.
     **                                                        *
     ** Return Codes:                                          *
     **    None                                                *
     **                                                        *
     ** Purpose:    Get status on the starting session.        *
     **                                                        *
     *************************************************************
     */

     void
     GetDispatchStatus(IN DD_getservicestatus_args *arg,
                       OUT DD_getservicestatus_result *res)
641  1  {
642  1      EDMSession    *sess;
643  1      EDMSession    *ret;
            static char buff[CONNECT_HANDLE_SIZE];

645  1      sess = new EDMSession();

647  1      if (sess == NULL)
648  2      { // Give an error
649  2          EDMDispatch_logent(
650  2                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                        "Failure to create a session block");
651  2          return;
652  1      }

654  1      sess -> setSessionID(&arg -> service_handle);

656  1      LockSessionMutex();

658  1      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

660  1      UnlockSessionMutex();

662  1      delete sess;

664  1      if (ret == NULL)
665  2      {
666  2          EDMDispatch_logent(
667  2                  __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
668  2                  "Failure to lookup session %ld:%ld",
                        arg -> service_handle.high,
                        arg -> service_handle.low);
670  2          res -> status = DD_SERVICE_FAILURE_NONEXEC;
671  2          return;
672  1      }

674  1      res -> status = ret -> getStatus();

676  1      memset(buff, 0, sizeof(buff));
```

```
678  1      if (res -> status == DD_SERVICE_RUNNING)
679  2      {
680  2          res -> handle.handle_val = (char *) ret -> getConnectionHandle(
                                                                            );
681  2          res -> handle.handle_len = CONNECT_HANDLE_SIZE;
682  1      }
683  1      else
684  2      {
685  2          res -> handle.handle_val = (char *) buff;
686  2          res -> handle.handle_len = CONNECT_HANDLE_SIZE;
687  1      }
688        }
```

```
690      /*********************************************
691      **
692      ** Routine:   GetDispatchInfo
693      **
694      ** Inputs:    DD_getservicestatus_args *arg - session ID to check the
695      **                                            status of
696      ** Outputs:   SessionBlock *res  -  the information regarding the
697      **                                  specified session
698      ** Return Codes:
699      **    None
700      **
701      ** Purpose:  Get status on all the sessions.
702      **
703      *********************************************
704      */
706      void
         GetDispatchInfo(IN DD_getservicestatus_args *arg,
707                      OUT SessionBlock *res)
708      {
709  1      EDMSession    *sess;
710  1      EDMSession    *ret;
711  1      SessionInfo   *sinfo, *slast;
712  1      SessionInfo   *ret;
713  1      static char buff[CONNECT_HANDLE_SIZE];

715  1      LockSessionMutex();

717  1      if (arg -> service_handle.high != 0 && arg -> service_handle.low != 0)
718  2      {
719  2          sess = new EDMSession();
720  2          // Looking for a single session. Do a find.

722  2          sess -> setSessionID(&arg -> service_handle);
723  2          ret = (EDMSession *) G_sessionTree.find(sess);

724  3          if (sess == NULL)
             {
                 // Give an error
                 EDMDispatch_logent(
                     __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                     "Failure to create a session block");
725  2              UnlockSessionMutex();
727  3              return;
728  2          }
729  2          sess -> setSessionID(&arg -> service_handle);
731  2          ret = (EDMSession *) G_sessionTree.find(sess);
733  2          delete sess;

735  3          if (ret == NULL)
             {
737  2              EDMDispatch_logent(
738  3                  __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
739  3                  "Failure to lookup session %ld:%ld",
740  3                  arg -> service_handle.high,
741  3                  arg -> service_handle.low);
742  3              UnlockSessionMutex();
743  3              return;
744  2          }
746  2          res -> totalsessions = 1;
```

```
748  2    res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

750  2    if (res -> sess == NULL)
751  3    {
752  3        EDMDispatch_logent(
753  3            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                     "Failure to allocate session info
                      block");

754  3
755  3        UnlockSessionMutex();
756  3        return;
         }

758  2    sinfo = res -> sess;

760  2    ret -> getSessionID(&sinfo -> service_handle);
761  2    sinfo -> status = ret -> getStatus();
762  2    sinfo -> jobstarttime = ret -> getStartTime();
763  2    sinfo -> operation_type = ret -> getOperationType();
764  2    sinfo -> lastSent = ret -> getLastSent();
765  2    sinfo -> lastReceived = ret -> getLastReceived();
     }
else
{

766  1
767  1
768  2
769  2    sinfo -> totalsessions = 0;

771  2    res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

773  2    if (res -> sess == NULL)
774  3    {
775  3        EDMDispatch_logent(
776  3            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                     "Failure to allocate session info
                      block");

777  3        UnlockSessionMutex();
778  2        return;
779  2    }

781  2    sinfo = res -> sess;

783  2    RWBinaryTreeIterator *sessionIterator = new
                     RWBinaryTreeIterator(G_sessionTree);

785  2    boolean_ty addnext = FALSE;

787  2    while ( sessionIterator != NULL && (ret = (EDMSession*) (
                     *sessionIterator)()) != NULL )
788  3    {
789  3        int             status;

791  3        if (addnext)
792  4        {
793  4            sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                         SessionInfo));
795  4            if (sinfo -> next == NULL)
796  5            {
797  5                break;
798  4            }

800  4            sinfo = sinfo -> next;
801  3        }

803  3        ret -> getSessionID(&sinfo -> service_handle);
804  3        sinfo -> status = ret -> getStatus();
805  3        sinfo -> jobstarttime = ret -> getStartTime();
```

```
806  3        sinfo -> operation_type = ret -> getOperationType();
807  3        sinfo -> lastSent = ret -> getLastSent();
808  3        sinfo -> lastReceived = ret -> getLastReceived();

810  3        getHandleSet(
811  3            &sinfo -> service_handle, &sinfo -> outhandle,
                     &sinfo -> errhandle, &status);

813  3        res -> totalsessions++;

815  3        sinfo -> next = NULL;
816  3        addnext = TRUE;
817  2    }

819  2    // through with iterator
820  2    if (sessionIterator != NULL)
821  3    {
822  2        delete sessionIterator;
823  2    }

825  1    }

827  1    UnlockSessionMutex();
828  1    }
```

```
830   /**********************************************************************
831   **
832   ** Routine:  removeSession
833   **
834   ** Inputs:
835   **
836   **
837   ** Outputs:
838   ** Return Codes:
839   **    None
840   **
841   ** Purpose: Remove the active session object between the GUI and the
842   **          Service.
843   **
844   */
846   int
847   removeSession(IN DD_client_session_id *sess_id,
848               OUT int *status)
849   {
850     EDMSession *sess;
851     EDMSession *ret;

853     if (status == NULL)
854     {
855       return -1;
856     }

858     if (sess_id == NULL)
859     {
860       *status = SESSION_BAD_ARGS;
861       return -1;
862     }

864     *status = 0;
865     if (G_sessionTree.isEmpty())
866     {
867       EDMDispatch_logent(
868          __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
869          "No sessions in list.
                  Can't remove session <%ld:%ld>",
                 sess_id -> high, sess_id -> low);
871       *status = SESSION_LIST_EMPTY;
872       return -1;
874     }

876     sess = new EDMSession();
877     if (sess == NULL)
878     {
879       EDMDispatch_logent(
880          __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
881          "Failure to create a session block");
882       return -1;
884     }

886     sess -> setSessionID(sess_id);

888     ret = (EDMSession *) G_sessionTree.remove(sess);
```

```
890     UnlockSessionMutex();

892     if (ret == NULL)
893     {
894       EDMDispatch_logent(
895          __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
896          "Failure to remove session %ld:%ld",
                 sess_id -> high, sess_id -> low);
897       delete sess;
898       *status = SESSION_LOOKUP_FAILED;
899       return -1;
900     }

902     delete ret;
903     delete sess;

905     return 0;
906   }
```

# D11

```
1   /***************************************************************
2   **
3   **
4   ** File Name:   RSLstart.c
5   **
6   ** Copyright (c) 1998,1999 by EMC Corporation.
7   **
8   ** Purpose:
9   ** --------
10  ** The intent of the contents of this file is to implement the
11  ** functions the control execution of the restore for the
12  **                  Restore Service
13  **                  Library.
14  **
15  ** These functions are provided to allow:
16  ** - creation of submit objects,
17  **     which define the set of objects to be
18  ** restored and the scripts to be run before and after
19  **                  restoration,
20  ** - starting the restoral of a submit object.
21  **
22  ** The following functions comprise restoral management:
23  **
24  ** RSTSL_Start
25  **
26  ** Compile-Time Options:
27  **     This section must list any compile time definitions
28  **     which will affect this header.
29  ****************************************************************/

30  /*
31   * Feature test switches.
32   * Standard defines required to turn on OS features go here.
33   *
34   * The following is required for code that uses POSIX API's.
35   * Remove for non-POSIX, non-portable code.
36   */
38  #define _POSIX_SOURCE 1

41  /*
42   * System headers.
43   */
45  #include <sys/wait.h>

47  /*
48   * Epoch headers.
49   */
50  #include <eb/eb_port.h>
51  #include <eb/rb_log.h>
52  #include <ebutil/eb_normalize.h>
53  #include <ebutil/ebutil.h>
54  #include <ebreport/ebvl.h>

57  /*
58   * Local headers
59   */
```

```
62   #include <RSLinterns.h>
63   #include <RSLauxSupp.h>
64   #include <restore/EDMRBSubmitApi.h>

66   #include <restore/REprogmsg.h>
67   #include <restore/dispatch_daemon.h>
68   #include <restore/EDMRBProgressApi.h>

70   extern int RunExecutable(const boolean_ty ResetUid,
71                            const int RunUid,
72                            const char *starting_cwd,
73                            const char *executable_name,
74                            char **executable_argv,
75                            char **executable_env,
76                            int *run_exit_status,
77                            boolean_ty *run_cancelled,
78                            boolean_ty (*QuitTest)(void));

80   extern int RunWorkItemRestores(int, boolean_ty (*CancelRestoreTest)());

82   static errno_ty
83   ExecuteWorkItemRestore(int SubmitObjectID,
84                          boolean_ty (*QuitTest)(void));

86   static errno_ty
87   RunPrepareRestore(int SubmitObjectID,
88                     boolean_ty (*QuitTest)(void),
89                     int *PrepareExit);

90   static errno_ty
91   RunCleanupRestore(int SubmitObjectID,
92                     boolean_ty (*QuitTest)(void),
93                     int runphase_status,
94                     int *CleanupExit);

96   /*
97    * #defines, structures, typedefs local to this source file
98    */

100  #define STR_SURE(str) (str) ? str:""
101  #define REMOVE_NEWLINE(str) \
102  {\
103      int rem_nl_index;\
104      for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
105      {\
106          if(str[rem_nl_index] == '\n')\
107              str[rem_nl_index] = '\0';\
108      }\
109  }

111  /****************************************************************
112   * Start
113   *
114   * This function begins execution of the restoral of the objects in a
115   * submit object.  Its progress and requests for operator input are
116   * returned via RSTSL_GetRestoreFeedback.
117   *
118   * Parameters:
119   *
120   * SubmitObjectID (
121   *     I) - ID of the submit object which describes the restore
122   * QuitTest
123   *     (I) - function to call to check for quit signal
124   ****************************************************************/
125  errno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(
                 void))
1    {
```

```
127 1   int ret_pre;
128 1   int ret_exec;
129 1   int ret_post;
130 1   int ret_all_ok;
131 1   int PrepareExit = 0;
132 1   int CleanupExit = 0;
133 1   boolean_ty QuitFlag = FALSE;

135 1   char asctime[32];

137 1   memset(asctime, 0, 32);

139 1   (void)time(&rcp->rc_cmd_starttime);

141 1   (void)ctime_r(&rcp->rc_cmd_starttime, asctime, 32);

143 1   rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;

145 1   REMOVE_NEWLINE(asctime);

147 1   rbe_log_stats(0, "Restore Started at %s.", asctime);

149 1   rbe_log_stats(0, "Restore Started application type %s.",
150 1       (rcp->rc_backup_app == 0)
151 1       ? "Network"
152 1       : ((struct pluginIDdata *)(
153 1           rcp-> currentPiptr-> idData))
                -> name ) ;

155 1   rbe_log_stats(0, "Restore Started of "
156 1       "top level object: %s, template %s, trailset %s.",
157 1       STR_SURE(rcp->rc_top_level_object_name),
158 1       STR_SURE(rcp->rc_template_name),
159 1       (rcp->rc_saveset_thread) ?"Alternate" : "Primary");

162 1   rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
163 1       STR_SURE(rcp->rc_human_uidname),
164 1       rcp->rc_human_uid, rcp->rc_human_gids[0]);

166 1   rbe_log_stats(0, "Restore Started with client destination %s.",
167 1       STR_SURE(rcp->rc_client_hostname));

169 1   /* if not a network restore,
            check if plugin has its own start function */

171 1   if ( rcp-> rc_backup_app != 0
172 1       && NULL != rcp-> currentPiptr-> piFuncArray[
                        PiFuncIndexStartRestore ] )
173 2   {
174 2       setGlobalStatus( EDMRE_STATE_EXECUTE );   /* set RE's internal status */
175 2       ret_exec = rcp-> currentPiptr-> piFuncArray[
                        PiFuncIndexStartRestore ]
176 2           (
                    rcp, SubmitObjectID, QuitTest);

178 2   if( QuitTest() )        /* check for abort before return */
179 3   {
180 3   rbe_log_stats( EP_RB_RECOVER_ABORT,
181 3       "The restore was quit by the user during
                execution.");
182 3   setGlobalStatus( EDMRE_STATE_USER_QUIT );   /* set RE's internal status */
```

```
183 3       return EP_RB_RECOVER_ABORT;
184 2   }

186 2   if (E_SUCCESS != ret_exec)
187 2       setGlobalStatus( EDMRE_STATE_FAILED );   /* set RE's internal status */
188 2   else
189 2       setGlobalStatus( EDMRE_STATE_SUCCESSFUL );   /* set RE's internal status */

191 2   }
192 1   return( ret_exec );

194 1   ret_pre = RunPrepareRestore(SubmitObjectID,
195 1       QuitTest,
196 1       &PrepareExit);

198 1   if(EP_RB_RECOVER_ABORT == ret_pre)
199 2   {
200 2   rbe_log_stats(EP_RB_RECOVER_ABORT,
201 2       "The restore was quit by the user during
                preparation.");
202 2   setGlobalStatus( EDMRE_STATE_USER_QUIT );   /* set RE's internal status */
203 2   return EP_RB_RECOVER_ABORT;
204 1   }

205 1   if(PrepareExit != 0)
206 2   {
207 2   rbe_log_stats(EP_RB_RECOVER_PREFAILED,
208 2       "The restore failed during preparation. Exit %d",
209 2       PrepareExit);

211 2   setGlobalStatus( EDMRE_STATE_FAILED );   /* set RE's internal status */
212 2   return (EP_RB_RECOVER_PREFAILED);
213 1   }

215 1   setGlobalStatus( EDMRE_STATE_EXECUTE );   /* set RE's internal status */

217 1   ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
218 1       QuitTest);

220 1   QuitFlag = Quittest();       /* check for abort before cleanup */

223 1   if (E_SUCCESS ==
224 2       ret_all_ok = ret_exec)   /* check if any WIs failed */
225 2   {
226 2   int local_stat;
227 2   EDMStats stats;
            memset( &stats, 0, sizeof(EDMStats) );

229 2   if (0 != getRestoreStatus( 0, &stats, &local_stat ))
230 3   {
231 3   rbe_log_stats(
232 3       "Internal error: Failed in getRestoreStatus.");
            ret_exec = ret_all_ok = EP_RB_RECOVER_EXECUTEFAILED;
233 2   }
234 2   else if (stats.edm.failed)
235 3   {   /* if any workitems failed,
                its a failure for cleanup purposes */
236 3   if (0 == stats.edm.successful)
237 3       ret_all_ok = EP_RB_RECOVER_ALLFAIL;
238 3   else if (stats.edm.successful > stats.edm.failed)
239 3       ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
240 3        else
241 3            ret_all_ok = EP_RB_RECOVER_MANYFAIL;
242 2
243 2        }
243 1    }
245 1    ret_post = RunCleanupRestore(SubmitObjectID,
246 1                                 QuitTest,
247 1                                 ret_all_ok,
248 1                                 &CleanupExit);
250 1    if(QuitFlag)
251 1    {
252 2        rbe_log_stats(EP_RB_RECOVER_ABORT,
253 2            "The restore was quit by the user during execution.");
254 2        setGlobalStatus( EDMRE_STATE_USER_QUIT );    /* set RE's internal status */
255 2        return EP_RB_RECOVER_ABORT;
256 1    }
258 1    if (E_SUCCESS != ret_exec)        /* return execute status if it failed */
259 2    {
260 2        setGlobalStatus( EDMRE_STATE_FAILED );    /* set RE's internal status */
261 2        return ret_exec;
262 1    }
264 1    if(EP_RB_RECOVER_ABORT == ret_post)
265 2    {
266 2        rbe_log_stats(EP_RB_RECOVER_ABORT,
267 2            "The restore was quit by the user during cleanup.");
268 2        setGlobalStatus( EDMRE_STATE_USER_QUIT );    /* set RE's internal status */
269 2        return EP_RB_RECOVER_ABORT;
270 1    }
272 1    if( (CleanupExit != 0)  ||  (E_SUCCESS != ret_post) )
273 2    {
274 2        rbe_log_stats(EP_RB_RECOVER_POSTFAILED,
275 2            "The restore failed during cleanup. Exit %d",
276 2            CleanupExit);
278 2        setGlobalStatus( EDMRE_STATE_FAILED );    /* set RE's internal status */
279 2        return (EP_RB_RECOVER_POSTFAILED);
281 1    }
283 1    setGlobalStatus( EDMRE_STATE_SUCCESSFUL );    /* set RE's internal status */
285 1    return( E_SUCCESS );
287 1    }    /* RSTSL_Start */
```

```
        static eerrno_ty
        ExecuteWorkItemRestore(int SubmitObjectID,
                               boolean_ty (*QuitTest)(void))
        {
292 1    int ret_RunWItem;
293 1    sm_push();
294      rcp->error_message[0] = 0;
297 1    if(0 != (ret_RunWItem = RunWorkItemRestores(
                                  SubmitObjectID, QuitTest)))
298 1    {
299 1        rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestores");
303 1        return EP_RB_RECOVER_EXECUTEFAILED;
304 2    }
305 2
306 1    if (ret_RunWItem != 0)
308 1        return EP_RB_RECOVER_ABORT;
310 1    if (QuitTest() == TRUE)
311 1        return EP_RB_RECOVER_ABORT;
313 1    sm_pop();
314 1
316 1    return E_SUCCESS;
317 1    }
```

```
320      #define EXECUTABLE_MAX 1024
321      static eerrno_ty
322      RunPrepareRestore(int SubmitObjectID,
323                        boolean_ty (*QuitTest)(void),
324                        int *PrepareExit)
325  1   {
326  1       char **prephaseargs = NULL;
327  1       char **prephaseenv = NULL;
328  1       int GetSOstatus = 0;
329  1       char preExecutable[EXECUTABLE_MAX];
330  1       boolean_ty restore_cancelled = FALSE;

332  1       *PrepareExit = 0;

334  1       /*
335  1        * GetSOPrePhase allocates prephaseargs & prephaseenv.
336  1        * This will need to be free'ed later.
337  1        *
338  1        */

340  1       if(0 != GetSOPrePhase(SubmitObjectID,
341  2                             preExecutable,
342  1                             EXECUTABLE_MAX,
343  1                             &prephaseargs,
344  1                             &prephaseenv,
345  1                             &GetSOstatus))
346  2       {
347  2           rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase");
348  2           return (EP_RB_RECOVER_FATALERR);
349  1       }

351  1       if(0 != strcmp(preExecutable, ""))
352  2       {
353  2           setGlobalStatus( EDMRE_STATE_PREPHASE );
354  2           if(-1 == RunExecutable(FALSE,
355  2                                  0,
356  2                                  NULL,
357  2                                  preExecutable,
358  2                                  prephaseargs,
359  2                                  prephaseenv,
360  2                                  PrepareExit,
361  2                                  &restore_cancelled,
362  2                                  QuitTest))
363  3           {
364  3               rbe_log_stats(
                        0,"Internal error: Failed in RunExecutable for prepare.");
365  3               return (EP_RB_RECOVER_FATALERR);
366  2           }
367  2           if(TRUE == restore_cancelled)
368  2               return (EP_RB_RECOVER_ABORT);
370  1       }

372  1       return( E_SUCCESS );
373      }
```

```
375      boolean_ty alwaysFalse() { return FALSE; }
```

```
377        static eerrno_ty
378  1     RunCleanupRestore(int SubmitObjectID,
379                boolean_ty (*QuitTest)(void),
380                int runphase_status,
381                int *CleanupExit)
382  1     {
384  1     char **postphaseargs = NULL;
385  1     char **postphaseenv = NULL;
386  1     int GetSOstatus = 0;
387  1     char postExecutable[EXECUTABLE_MAX];
388  1     boolean_ty restore_cancelled = FALSE;
389  1     boolean_ty ignore_quit=FALSE;

391  1     *CleanupExit = 0;

393  1     /*
394  1      * GetSOPostPhase allocates postphaseargs & postphaseenv.
395  1      * This will need to be free'ed later.
396  1      *
397  1      */
398  1     PurgeTrailQueue();
399  1     if(0 != GetSOPostPhase(SubmitObjectID,
400  1                 postExecutable,
401  1                 EXECUTABLE_MAX,
402  1                 &postphaseargs,
403  1                 &postphaseenv,
404  1                 &GetSOstatus))
405  2     {
406  2     rbe_log_stats(0,"Internal error: Failed in GetSOPostPhase");
407  2     return (EP_RB_RECOVER_FATALERR);
408  1     }

410  1     #define RESTORE_BREAK        "RESTORE_BREAK="
411  1     #define RESTORE_BREAK_TRUE   "RESTORE_BREAK=T"
412  1     #define RESTORE_BREAK_ERROR  "RESTORE_BREAK=E"

414  1     if(0 != strcmp(postExecutable, ""))
415  2     {
416  2      /*
417  2       * If a quit has been specified, we need to tweak the
418  2       * RESTORE_BREAK environment variable if set
419  2       */
420  2     char *abort=NULL;
421  2     if(QuitTest())
422  3     {
423  3     abort=RESTORE_BREAK_TRUE;
424  3     }
425  2     else if(0!=runphase_status)
426  3     {
427  3     abort=RESTORE_BREAK_ERROR;
428  2     }

430  2      /*
431  2       * ignore_quit is set to 1 when we have already processed a BREAK
432  2       * (CANCEL from gui), and are using the environment variable
433  2       * RESTORE_BREAK_E to signal the post restore script to clean
434  2       * up form this break.  When that happens, an ignore_quit value
435  2       * of 1 will cause actual quit signals to be ignores by the
436  2       * cleanup script, since we already know (via the environment
437  2       * variable) that we are in "cleanup mode" and no further signal
438  2       * interception is necessary.
439  2       */
440  2     ignore_quit=FALSE;
441  2     if(NULL!=abort && NULL!=postphaseenv)
```

```
442  3     {
443  3     int isub=0;
444  3     char *cptr;
445  3     while(cptr=postphaseenv[isub])
446  4     {
447  4     if(strcmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                 RESTORE_BREAK))==0)
448  5     {
449  5     postphaseenv[isub]=esl_strdup(abort);
450  5     ignore_quit=TRUE;
451  5     if(NULL==postphaseenv[isub])
452  6     {
453  6     rbe_log_stats(
454  6     EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RSLstart.c");
455  6     return EP_RB_RECOVER_POSTFAILED;
456  5     }
457  4     }
458  3     isub++;
459  2     }
461  2     setGlobalStatus( EDMRE_STATE_POSTPHASE );   /* set RE's internal status */
462  2     if(-1 == RunExecutable(FALSE,
463  2                 0,
464  2                 NULL,
465  2                 postExecutable,
466  2                 postphaseargs,
467  2                 postphaseenv,
468  2                 CleanupExit,
469  2                 &restore_cancelled,
470  2                 ignore_quit?alwaysFalse:QuitTest))
471  3     {
472  3     rbe_log_stats(
473  3     0,"Internal error: Failed in RunExecutable for cleanup.");
474  2     return (EP_RB_RECOVER_FATALERR);
475  2     }
476  2     if(TRUE == restore_cancelled)
           return (EP_RB_RECOVER_ABORT);
478  1     }
479  1     return( E_SUCCESS );
480  1     }
```

```
482    static eerrno_ty
483    RunExecutionOverrideRestore(int SubmitObjectID,
484                               boolean_ty (*QuitTest)(void))
485  1 {
487  1    return( E_SUCCESS );
488    }
```

```
489    #undef EXECUTABLE_MAX
```

```
1   /*********************************************************
2   **                                                   *******
3   ** File Name:   RSLwisvr.c
4   **
5   ** Copyright (c) 1998,1999 by EMC Corporation.
6   **
7   ** Purpose:
8   ** --------
9   ** The intent of the contents of this file is to implement the
10  ** functions the control execution of work item restores for
                                                            Restore
11  **          Service Library.
12  **
13  **     These functions are provided to allow:
14  **
15  ** The following functions comprise restoral management:
16  **
17  **     RunWorkItemRestores()
18  **
19  **
20  ** Compile-Time Options:
21  **     This section must list any compile time definitions
22  **     which will affect this header.
23  **
24  *********************************************************
                                                    *****/

27  #define _POSIX_SOURCE 1

30  /*
31   * System headers.
32   */

34  #include <sys/time.h>
35  #include <sys/types.h>
36  #include <sys/wait.h>
37  #include <values.h>

39  /*
40   * Epoch headers.
41   */
42  #include <eb/eb_port.h>
43  #include <eb/rb_log.h>
44  #include <ebutil/ebutil.h>
45  #include <restore/REprogmsg.h>

47  /*
48   * Local headers
49   */

51  #include <RSLspexits.h>
52  #include <RSLremfd.h>
53  #include <EDMRESchedApi.h>
54  #include <EDMREHandleMgrApi.h>
55  #include <RSLinterns.h>
56  #include <RSLrbmain.h>
57  #include <restore/EDMRESubmitApi.h>
58  #include <EDMREDrainApi.h>

60  #define STR_SURE(str) (str) ? str:""

63  /*
```

```
64   *
65   * RunWorkItemRestores
66   * {
67   * Set the number of drives being used for the life of the restore.
68   * SetQuitFlag = FALSE
69   * TrailRestoresLeft = ( # of trail restores )
70   * TrailRestoresRunning = 0;
71   *
72   * while drive available.
73   *     RunTrail -- Set drive concurrency for trail restore.
74   *     TrailRestoresRunning++;
75   *     while OKToRunWIForTrail
76   *         StartWIRestore
77   *
78   * while(1)
79   *     if(QuitTest)
80   *         Send WICancells
81   *         SetQuitFlag = TRUE
82   *     Select(from_pipe, timeout (5 seconds))
83   *     for each WI that completes
84   *         interperate return.
85   *         Drain progress.
86   *         Send final Progress for work item.
87   *         if(WIfailed)
88   *             if(OKtoReschedule && SetQuitFlag == FALSE)
89   *                 Add to TrailQueue
90   *         if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
91   *             while OKToRunWIForTrail
92   *                 StartWIRestore
93   *         else -- RunNewTrailRestore
94   *             EndTrailRestore(prevTrailQueue)
95   *             TrailRestoresRunning--;
96   *             while (drive available &&
97   *                     SetQuitFlag == FALSE &&
98   *                     TrailRestoresLeft)
99   *
100  *                 RunTrail -- Set drive concurrency for trail restore.
101  *                 TrailRestoresRunning++;
102  *                 while (
103  *                     OKToRunWIForTrail && SetQuitFlag == FALSE)
104  *                     StartWIRestore
105  *                 end while
106  *             end while
107  *     End for each WI completes
108  *     End else
109  *     if (((SetQuitFlag == TRUE) && (TrailRestoresRunning == 0)) ||
110  *         (TrailRestoresLeft == 0)
111  *         return; exit loop.
112  * end while(1)
113  *
114  * }
115  */
116
118  /* Stubs */
119  static int
120  InterpretWorkItemRestoreResults(wi_restore_results *results);
122  static int
123  test_fd(int fd);
125  static void
126  DebugLogFds(char *error_msg,
127                  fd_set *fds);
```

```
129
130   static int
      DetermineGlobalDriveUse();
132
133   static int
      test_fd_hup(int fd);
135
136   static int
137   FindTrailIDForWItem(int handle,
138                       int *TrailID,
                          int *status);
140
141   static int
      SendRunningWorkItemsQuit();
143
      /* End Stubs */
145
146   static int
147   Select(int nfds,
148          fd_set *readfds,
149          fd_set *writefds,
150          fd_set *exceptfds,
           struct timeval *timeout);
153
154   static int
      HandleWorkItemRestoreResults(int FromFD,
155                                 int *TrailID,
156                                 wi_restore_results *results);
157
158   static int
159   RunWorkItemRestoresForTrail(const int TrailID,
160                               const int CountDrivesAvailable,
161                               boolean_ty (*CancelRestoreTest)(),
162                               boolean_ty *QuitFlag,
                                 int *CountDrivesInUse);
164   /*
165    *  RunWorkItemRestores()
166    *
167    *  Runs a set of work item restores.
168    *
169    *  Args:
170    *  SubmitObject
171    *  CancelRestoreTest().
172    *
173    *  Returns: int 0 for success.
174    *
175    */
176   int
      RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)()
177 1 {
178 1     boolean_ty QuitFlag = FALSE;      /* Has the user requested a quit.*/
179 1     boolean_ty SentQuit = FALSE;      /* Have we initiated the quit.*/
181 1     int TrailRestoresRunning = 0;     /* The number of trail restores running. */
182 1     int TrailRestoresLeft;            /* The number of trail restores left. */
183 1     int TrailRestoresTotal;           /* The number of trail restores total. */
185 1     int CountDrivesAvailable;         /* The count of drives available. */
186 1     int CountDrivesInUse = 0;         /* The count of drives in use.*/
188 1     int temp_status;
189 1     int HighestActiveTrail = 0;       /* The trail queues are ordered from 1 to n. */
```

```
190 1
192 1     if(debugmode)
193 1     {
194 2        (void)rbe_user_error(0,
195 2            "DEBUG: Running RunWorkItemRestores.");
196 1     }
197 1     /* GenerateTrailQueues()
198 1      * Buckets the work items into trail queues.
199 1      *      The trail queues are sorted
200 1      * in the order which the restores should run.
          */
202 1     if(0 != GenerateTrailQueues(SubmitObject,
203 1                                 &TrailRestoresTotal,
204 1                                 &temp_status))
205 1     {
206 2        (void)rbe_user_error(0,
207 2            "Internal error: Cannot generate trail
                   queues, cannot continue.");
208 2        return -1;
209 1     }
211 1     TrailRestoresLeft = TrailRestoresTotal;
213 1     CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);
215 1     if(debugmode)
216 1     {
217 2        (void)rbe_user_error(0,
218 2            "DEBUG: RunWorkItemRestores for %d trails.",
219 2            TrailRestoresTotal);
220 1     }
221 1     /*
222 1      * This is the start up loop to get the initial work item
223 1      * restores started.
224 1      */
226 1     QuitFlag = CancelRestoreTest();
228 1     while((CountDrivesInUse < CountDrivesAvailable) &&
229 1           (HighestActiveTrail < TrailRestoresTotal) &&
230 1           (FALSE == QuitFlag))
231 2     {
232 2        int submitObjID = 0;
233 2        int submitelementID = 0;
235 2        HighestActiveTrail++;
237 2        /*
238 2         * Activate the Trail Queue.
239 2         *      This allows the trailid queues to be used to
240 2         * determine the work item restores to run.
241 2         */
242 2        if(0 != ActivateTrailQueue(HighestActiveTrail,
243 2                                   1,
244 3                                   &temp_status))
245 3        {
246 3           (void)rbe_user_error(0,
247 3               "Internal error: Cannot activate trail
                      queues(1) for trailid %d, cannot continue.",
248 3               HighestActiveTrail);
249 2           return -1;
           }
```

```
251  2  	/*
252  2  	 * This sets the number of drives and media access concurrency for
         	 	trail restores.
253  2  	 * i.e. The count of running work item restores for this trail.
         	 	Today this is one.
254  2  	 */
255  2  	if(0 != SetQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
256  2  	{
257  3  		(void)rbe_user_error(0,
258  3  			"Internal error: Cannot set drive acquired(
259  3  			1) for trailid %d, cannot continue.", HighestActiveTrail);
260  2  		return -1;
262  2  	}
263  3
264  2  	if (0 > (temp_status = RunWorkItemRestoresForTrail(
265  2  		HighestActiveTrail,
266  2  		CountDrivesAvailable,
267  3  		&CountDrivesInUse))
269  3  	{
270  3  		/* RunWorkItemRestoresForTrail does its own error logging. */
271  2  		return -1;
273  2  	}
274  3
275  3  	if(temp_status == 0)
276  3  	{
277  3  		(void)rbe_log_stats(0,
278  2  			"Trail %d restore had no work item to run(
279  2  			1).", HighestActiveTrail);
280  3
281  3  		/* more work may be need to recover from this error condition. */
282  2  	}
283  1
285  1  	if(temp_status > 0)
286  2  	{
287  2  		TrailRestoresRunning++;
288  2  	}
289  2
290  2  	} /* End while() initial startup loop */
292  2  	while(1)
293  3  	{
294  3  		int HighestFd = 0;
295  3  		fd_set WorkItemFromFds;
297  3  		int retStatus;
298  2  		struct timeval timeout = (5, 0);
299  2
301  2  		if((QuitFlag) && (!SentQuit))
302  3  		{
303  3  			(void)rbe_log_stats(0,
304  3  				"Restore was quit by user. Quitting restore,
                 			this could take a while.");

            			SendRunningWorkItemsQuit();
            			SentQuit = TRUE;
            		}

            		if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
            		{
            			(void)rbe_user_error(0,
            				"Internal error: Cannot get auxproc result
            				fds, cannot continue.");
```

```
306  3  			return -1;
307  2  		}
308  2  	}
309  2  #if 0
310  3  		if(debugmode)
311  3  		{
312  3  			DebugLogFds("The file descriptors to wait on are ",
             				&WorkItemFromFds);
313  2  		}
314  2  #endif
315  2  		if(0 > (retStatus = Select(HighestFd + 1,
316  2  			&WorkItemFromFds,
317  2  			NULL, NULL,
318  2  			&timeout)))
319  2  		{
320  3  			/* error */
321  3  			(void)rbe_user_error(RRECOVER_MKERR(errno),
323  3  				"Internal error: Cannot get auxproc result
324  2  				fds, cannot continue.");
326  2  			return -1;
327  3  		}
329  3  		QuitFlag = CancelRestoreTest();
331  2  	}
333  3  	else if (0 == retStatus)
334  3  	{ /* timed out */
336  3  	}
337  3
338  3  	else
340  3  	{ /* Available fds */
341  4  		int ReadyFds = retStatus;
342  4  		int FoundFds = 0;
343  4  		int index;
344  4
346  3  		if(debugmode)
347  3  		{
348  3  			DebugLogFds("The file descriptors ready to read are ",
349  3  				&WorkItemFromFds);
350  3  		}
351  3
353  3  		/*
354  3  		 * If there are available fds then we may want to
355  3  		 * schedule the next work item restore. We should
356  3  		 * check if the user initiated a quit.
357  4  		 */
358  4  		QuitFlag = CancelRestoreTest();
359  5  		for(index = 0;
360  5  			(index < (HighestFd + 1)) && (FoundFds < ReadyFds);
361  5  			index++)
362  5  		{
363  5  			int StartWorkItemforTrail = 0;
365  5  			if(FD_ISSET(index, &WorkItemFromFds))
367  5  			{
368  5  				int TrailID;
369  5  				int TrailAcquired;
                 				wi_restore_results results;
                 				FoundFds++;

                 				memset(&results, 0, sizeof(wi_restore_results));
370  6  				if(0 != HandleWorkItemRestoreResults(index,
                 					&TrailID,
                 					&results))
```

```
371  6        /* HandleWorkItemRestoreResults will do its own logging! */
372  6
373  5        }
374  5        return -1;
375  5        /*
376  5         * This is where we may want to retry the Work Item
377  5         * Based on if it passes or fails
              */
379  5        CountDrivesInUse--;

382  5        if (0 > (StartWorkItemforTrail =
383  5            RunWorkItemRestoresForTrail(TrailID,
384  5                CountDrivesAvailable,
385  5                CancelRestoreTest,
386  5                &QuitFlag,
387  5                &CountDrivesInUse)))
388  6        {
389  6            /* RunWorkItemRestoresForTrail does its own logging. */
390  5            return -1;
391  5        }

393  5        else if (StartWorkItemforTrail == 0)
394  5        {
395  5            /* 0 work items started above,
396  5             * lets check to see if this is the last work item
                      left for
397  5             * this trail
                  */
398  6            int wiCount;

399  6            if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
401  6            {
402  7                (void)rbe_user_error(0,
403  7                    "Internal error: Cannot determine
              number of running work items for trail, cannot continue.");
404  7                return -1;
406  7            }

407  6            if (debugmode)
408  6            {
409  7                (void)rbe_user_error(0,
410  7                    "DEBUG: RunWorkItemRestores no
              more work items left for trailid %d ,
              but %d wiCount workitem still running.",
              TrailID, wiCount);
411  7            }

412  7            /* Testing for No work items left running or started
413  6             * For this trail.
              */
414  6            if((0 == wiCount) && (0 == StartWorkItemforTrail))
415  6            {
416  6                TrailRestoresRunning--;
418  6                TrailRestoresLeft--;

419  7                if(0 != DeactivateTrailQueue(TrailID, &temp_status))
421  7                {
422  7                    (void)rbe_user_error(0,
424  7                        "Internal error: Cannot
              deactivate trail queue for trailid %d, cannot continue.", TrailID);
425  8                    return -1;
426  8                }
```

```
432  7        /* This test is to determine, if a trail restore finished,
433  7         * there may be another not yet started trail, If we have
434  7         * drive available the next trail restore will be
435  7         * started.
              */
436  7
437  7        if ((0 != TrailRestoresLeft) &&
438  7            (HighestActiveTrail < TrailRestoresTotal) &&
439  8            (CountDrivesInUse < CountDrivesAvailable))
441  8        {
443  8            HighestActiveTrail++;
444  8            if(0 != ActivateTrailQueue(HighestActiveTrail,
445  8                1,
446  9                &temp_status))
447  9            {
448  9                (void)rbe_user_error(0,
449  9                    "Internal error: Cannot
              activate trail queue(2) for trailid %d, cannot continue.",
450  9                    HighestActiveTrail);
451  8                return -1;
              }
453  8            if(0 != SetTQDrivesAcquired(
454  9                HighestActiveTrail, 1, &temp_status))
455  9            {
456  9                (void)rbe_user_error(0,
457  9                    "Internal error: Cannot set
              drive acquired(2) for trailid %d, cannot continue.",
458  9                    HighestActiveTrail);
459  8                return -1;
              }
461  8            if (0 > (temp_status = RunWorkItemRestoresForTrail(
462  8                HighestActiveTrail,
463  8                CountDrivesAvailable,
464  8                CancelRestoreTest,
465  8                &QuitFlag,
466  9                &CountDrivesInUse)))
467  9            {
468  9                /* RunWorkItemRestoresForTrail does its own logging. */
469  9                return -1;
470  8            }
471  9            if(temp_status == 0)
472  9            {
473  9                /* If this Trail Had no work items we
474  9                 * Should attempt to run the next trails
475  9                 * work items here. This would be an internal
476  9                 * error if a trail queue had no work item
477  9                 * restores.
478  9                 */
479  9                (void)rbe_log_stats(0,
480  8                    "Internal error: Trail %d
              restore had no work item to run(1).", HighestActiveTrail);
481  8                return -1;
482  8            }
483  9            if(temp_status > 0)
              {
```

```
484  9          /* If at least on work item was started for this trail,
485  9           * then we have started a new trail.
486  9           */
487  8          TrailRestoresRunning++;
488  8        }
489  7      }
491  6    }
492  5  }
493  4  } /* end for() */
494  3  } /* else Available fds */

496  2  /*
498  2   * Terminate the loop if either
499  2   * --------------------------------
500  2   * 1) Sent the work items the quit AND
501  2   *    No Trail restores a running.
502  2   * OR
503  2   * 2) No more Trail restores are left.
504  2   */
505  2  if(((0 == TrailRestoresRunning) && (SentQuit)) ||
507  2     (0 == TrailRestoresLeft))
508  2  {
509  3    break;
510  3  }
511  2  } /* end while(1) */
513  1  if((0 == TrailRestoresRunning) && (SentQuit))
514  1  {
515  2    (void)rbe_log_stats(0,
516  2          "Restore was quit by user.
517  2           Work item restore quit.");
518  1  }
519  1  return 0;
520     }
```

```
522  1  /* Functions needed
523     SendRunningWorkItemsQuit();
524     InterpretWorkItemRestoreResults();
525     */

527     static int
528     Select(int nfds,
529  2         fd_set *readfds,
530  2         fd_set *writefds,
531  2         fd_set *exceptfds,
532  2         struct timeval *timeout)
533  1  {
534  1    int retSelect;

536  1    do
537  2    {
538  2      retSelect = select(nfds,
539  2                         readfds,
540  2                         writefds,
541  2                         exceptfds,
542  2                         timeout);
544  1    } while ((-1 == retSelect) && (EINTR == errno));

546  1    return retSelect;
547  1  }
```

```
550
551  eperrno
552  InitiateWorkItemRestore(const int SubmittObjID,
553                          const int SubmitElemID)
554 1 {
555 1   struct auxproc AuxprocVitals;
556 1   eerrno_ty StartupAPResults = EXIT_FAILURE;
557 1   time_t StartTime;
558 1   int TempStatus;
559 1   char junk_executable[1024];
560 1   char **junk_argv;
561 1   char **AP_env = NULL;
562 1   int SOstatus;
563 1   char clientName[256] = "";
564 1   int clientPort;
565 1   int status;
566 1   time_t EndTime;
567 1   /* Lets see if there are any environment variables to set.
568 1    * The restore of the output variables are ignored.
569 1    */
570 1   if(E_SUCCESS != GetSOExecutionPhase(SubmittObjID,
571 1                       junk_executable, 1024,
572 1                       &junk_argv,
573 1                       &AP_env,
574 1                       &SOstatus))
575 2   {
576 2     (void)rbe_user_error(0,
577 2         "Internal Error: Could not get environment
                  variables.");
578 2     return -1;
        }
580 1   if (E_SUCCESS == GetSERcmdConnect(SubmittObjID,
581 1                       SubmitElemID,
582 1                       clientName, 256,
583 1                       &clientPort,
584 1                       &SOstatus))
585 2   {
586 2     StartupAPResults = StartupAuxprocess(0 /* XXX */,
587 2                       &AuxprocVitals,
588 2                       AP_env,
589 2                       clientName,
590 2                       clientPort);
591 2   }
592 2   else
593 2   {
594 2     (void)rbe_user_error(0,
595 2         "Internal Error: Could not get Remote Client name
                  &"
596 2         "port to connect.");
597 1     return -1;
598 1   }
600 1   if(E_SUCCESS != StartupAPResults)
601 2   {
602 2     /* StartupAuxprocess does its own logging. */
603 2     return -1;
604 1   }
606 1   /*
607 1    * We need to close the bulk fd. This file descriptor
608 1    * is not being used any more. If we do not close it
609 1    * here we will have a file descriptor leak because
610 1    * we won't be able to determine what it was when the
611 1    * work item completes.
```

```
612 1    */
614 1    close(AuxprocVitals.xp_fd_bulk_to_x);
616 1    time(&StartTime);
618 1    if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
619 1                         AuxprocVitals.xp_fd_from_x,
620 1                         AuxprocVitals.xp_fd_prog_from_x,
621 1                         SubmittObjID,
622 1                         SubmitElemID,
623 1                         AuxprocVitals.xp_pid,
624 1                         StartTime,
625 1                         &TempStatus))
626 2    {
627 2      (void)rbe_user_error(
628 2          0, "Internal Error: Could not register handle set.");
629 1      return -1;
         }
632 1    if(0 > StartWorkItemRestore(tcp,
633 1                         &AuxprocVitals,
634 1                         SubmittObjID,
635 1                         SubmitElemID))
636 2    {
637 2      /*
638 2       * StartWorkItemRestore does logging if initialization fails
639 2       */
640 2      (void)rbe_user_error(
641 2          0, "Error in StartWorkItemRestore  SubmittObjID %d, "
642 2              "SubmitElemID %d ", SubmittObjID,
643 2                   SubmitElemID);
644 2      /*
645 2       * the following code kills auxproc when recx or xcpio do not
                   start
646 2       * we do not want an auxproc sitting around.
647 2       * if errors occur in deleteHandleSet or KillWorkItemRestore the
                   messages are
648 2       * logged in those calls, plus
                   we already know there was an error and that
649 2       * is why we are doing this right now.
             */
651 2      deleteHandleSet(
                   AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL, &status);
653 2      KillWorkItemRestore(
                   AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);
655 2      time(&EndTime);
656 1      return -1;
         }
658 1    return 0;
660 1  } /* InitiateWorkItemRestore() */
```

```c
663  /*
664   * interperate return.
665   * Drain progress.
666   * Send final progress for work item.
667   * Delete the handle set.
668   */

670  static int
671  HandleWorkItemRestoreResults(int FromFD,
672                               int *TrailID,
673                               wi_restore_results *results)
674  {
675      int ret = 0;
676      int retries = 0;
677      int GetAuxprocResultsStatus;
679      int TempStatus;
680      int DrainResult;
681      int DrainedFD;
682      int wiCount;
683      int AuxProcPid;
684      int ToFD, getFromFD, ProgressFD;
685      time_t EndTime;
686      unsigned long jobstat;
687      int timeout = 3; /* Lets try 3 seconds */
         boolean_ty fromFDHangUp = FALSE;

689      ToFD = getFromFD = ProgressFD = -1;

691      while(! (fromFDHangUp))
692      {
693          GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

695          if(-1 == GetAuxprocResultsStatus)
696          {
                 /* GetAuxprocResults() does its own logging */
697              (void)rbe_user_error(0," Error in GetAuxprocResults");
699              return -1;
700          }

701          if(0 == GetAuxprocResultsStatus)
702          {
703              if(test_fd_hup(FromFD) == 1)
704              {
705                  fromFDHangUp = TRUE;
706              }

708              /* The remote result are not always going to
                  * be set. For example if the remote command
                  * is not started correctly.
                  */
710              if(results->local_exit_set == TRUE)
711              {
712                  break;
713              }
715              else
716              {
717                  sleep(1);
718                  test_fd(FromFD);
719                  continue;
720              }
721          }
```

```c
727      time(&EndTime);

729      if(0 != PushDrainRequest(FromFD, &TempStatus))
730      {
731          (void)rbe_user_error(0,
                 "Internal error: Could not push drain
                 request, cannot continue.");
732          return -1;
734      }

         /* Lets give the progress thread a chance to drain keeping busy in
          * the meanwhile.
          */

740      if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
741      {
742          (void)rbe_user_error(0,
                 "Internal error: Could not find trail id for
                 finished work item, cannot continue.");
743          return -1;
745      }

748      if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
749      {
750          (void)rbe_user_error(0,
                 "Internal error: Could not decrement running
                 work items for trail, cannot continue.");
751          return -1;
753      }

756      if(0 != getPID(FromFD, &AuxProcPid, &TempStatus))
757      {
758          (void)rbe_user_error(0,
                 "Internal error: Could not get auxproc pid
                 for work item, cannot continue.");
759          return -1;
761      }

764      if(0 != getHandleSet(
                 FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
765      {
766          (void)rbe_user_error(0,
                 "Internal error: Could not get auxproc file
                 descriptors for work item, cannot continue.");
767          return -1;
769      }

771      if(FromFD != getFromFD)
772      {
773          (void)rbe_user_error(0,
                 "Internal error: mismatch on from file
                 descriptors for work item, cannot continue.");
775          return -1;
776      }

778      while (0 != (ret = PopDrainResult(timeout,
779                      &DrainResult,
780                      &DrainedFD,
781                      &TempStatus)) && retries < 3)
782      {
784          if (ret != 0)
785          {
                 retries++;
```

```
786  2        (void)rbe_user_error(0,
787  2            "Internal error: Could not pop drain results,
                   cannot continue.");
789  2        }
790  1
792  1    return -1;
793  1
794  1    /*
796  1     *    Send final Progress for work item. XXX
797  1     *
798  1     */
800  1    /* Translate the local and remote error statuses
801  1     * to an eperrno value:
802  1     */
803  2    if(0 != results->local_exit_status)       /* use local error, if any */
804  2    switch (results->local_exit_status)
805  2    {
806  2    case XG_EXIT_ALLFAIL:
807  2        jobstat = EP_RB_RECOVER_ALLFAIL;
808  2        break;
809  2    case XG_EXIT_MANYFAIL:
810  2        jobstat = EP_RB_RECOVER_MANYFAIL;
811  2        break;
812  2    case XG_EXIT_FEWFAIL:
813  2        jobstat = EP_RB_RECOVER_FEWFAIL;
814  2        break;
815  2    case SPEXIT_REMOTE_STDERR_PROTOCOL:
816  2    case SPEXIT_REMOTE_STDERR_FAIL:
817  2        jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
818  2        break;
819  2    case XG_EXIT_STOPPED:         /* treat like signal */
820  2    default:         /* check for signal termination vs all generic
                                                      failures */
817  2    if ( XG_EXIT_SIGBASE < results->local_exit_status
819  2       || XG_EXIT_STOPPED == results->local_exit_status )
820  3    { /* killed by signal or stopped: separate error for sigpipe */
821  3    if (XG_EXIT_SIGBASE + SIGPIPE == results->local_exit_status)
822  3        jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
823  3    else
824  3        jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
825  2    }
826  2    else
827  2    { /* generic server failure, unless client failed too */
828  3    jobstat = EP_RB_RECOVER_SERVERFAIL;
829  3    if (0 != results->remote_exit_status)
830  3        jobstat = EP_RB_RECOVER_BOTHFAIL;
831  2    }
832  2    }
833  1    else if(0 != results->remote_exit_status)
834  1    jobstat = EP_RB_RECOVER_CLIENTFAIL;
835  1    else
836  1    jobstat = E_SUCCESS;
838  1    if( (0 != results->remote_exit_status) ||
839  1        (0 != results->local_exit_status ) )
840  1    {
841  2    int    status=0;
842  2    int    rc=0;
843  2    char   *templateName=NULL;
844  2    char   *winame=NULL;
845  2    char   *trailsetName=NULL;
847  2    rc = getHandleSetInformation(FromFD,
848  2                                 &templateName,
849  2                                 &winame,
```

```
850  2                                 &trailsetName,
851  2                                 &status);
852  2    rbe_log_stats(0, "Restore Failure of \"
853  2        "top level object: %s, template %s.",
854  2        STR_SURE(winame),
855  2        STR_SURE(templateName));
856  2    free(templateName);
857  2    free(winame);
858  2    free(trailsetName);
859  1    }
861  1    if( 0 != deleteHandleSet(FromFD, EndTime, jobstat, &TempStatus))
862  2    {
863  2    (void)rbe_user_error(0,
864  2        "Internal error: Could not delete Handle
                                 Set, cannot continue.");
866  2    return -1;
867  1    }
869  1    if(0 != KillWorkItemRestore(AuxProcPid,
870  1                    -1 /* Hack this arg is not needed yet
                                                      cmd_to */)
871  2    {
872  2    (void)rbe_user_error(0,
873  2        "Internal error: Could not kill finished
                         auxproc, cannot continue.");
874  2    return -1;
875  1    }
877  1    close(ToFD);
878  1    close(FromFD);
879  1    close(ProgressFD);
881  1    if(debugmode)
882  2    {
883  2    (void)rbe_user_error(0,
884  2        "DEBUG: HandleWorkItemRestoreResults Auxproc(
                      PID %d) just finished for trailid %d work items left = %d.",
885  2        AuxProcPid,
886  2        *TrailID,
887  2        wiCount);
890  2    (void)rbe_user_error(0,
891  2        "DEBUG: HandleWorkItemRestoreResults Auxproc(
                      PID %d) results are local: %d  setp:%s remote: %d set:%s.",
892  2        AuxProcPid,
893  2        results -> local_exit_status,
894  2        results -> local_exit_set) ? "TRUE" : "FALSE" ,
895  2        results -> remote_exit_status,
898  1        results -> remote_exit_set) ? "TRUE" : "FALSE");
899  1    }
900  1    return 0;
          } /* End HandleWorkItemRestoreResults() */
```

```c
904   /*
905    *
906    *  RunWorkItemRestoresForTrail()
907    *
908    *  Description
909    *   This function starts all the work item for the
910    *   trail. For no this is set to one but concurrency
911    *   will can be supported.
912    *
913    *  Args:
914    *   (I) TrailID -- The id for this trail.
915    *   (I) CountDrivesAvailable -- the total drives available to restore.
916    *   (O) QuitFlag -- indicatate whether the user has quit the restore.
917    *   (O) CountDrivesInUse -- The count of trails in use by restore.
918    *
919    *  Return int
920    *   if -1 then an error has occurred.
921    *   if 0 or greater then the number of trail restores started will be
922    *   returned.
923    *
924    */
925   static int
      RunWorkItemRestoresForTrail(const int TrailID,
                                  const int CountDrivesAvailable,
                                  boolean_ty (*CancelRestoreTest)(),
                                  boolean_ty *QuitFlag,
                                  int *CountDrivesInUse)
930   {
931     int  DriveAcquiredForTrail;
932     int  DriveConcurrencyForTrail;
933     int  submitObjID;
934     int  submitelementID;
935     int  popResults = 0;
936     int  temp_status;
937     int  CountOfWorkItemRestoresStarted = 0;
938     int  wiCount;

940     while(1)
941     {

944       (*CountDrivesInUse)++;

946       if(0 != (popResults = PopWIFromTrailQueue(TrailID,
947                                                 &submitObjID,
948                                                 &submitelementID,
949                                                 &temp_status)) &&
950                              (SCHED_NO_MORE_JOBS != temp_status))
951       {
952         (void)rbe_user_error(0,
953           "Internal error: Cannot pop work item off
954            trail queue, cannot continue.");

954         return -1;
955       }
957       if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
958       {
959         return CountOfWorkItemRestoresStarted;
960       }

962       temp_status = InitiateWorkItemRestore(
                          submitObjID, submitelementID);

964       if(temp_status != 0)
```

```c
965       {
966         /* InitiateWorkItemRestore() does its own logging */
967         (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,"
968           "submitObjID %d, submitelementID %d", submitObjID,
                  submitelementID);
969         return -1;
970       }

972       if(0 != IncrementRunningWI(TrailID, &wiCount, &temp_status))
973       {
974         (void)rbe_user_error(0,
975           "Internal error: Could not increment
976            running work items for trail, cannot continue.");
977         return -1;
979       }
          CountOfWorkItemRestoresStarted++;

981       if(0 != GetTQDrivesAcquired(TrailID,
982                                   &DriveAcquiredForTrail,
983                                   &temp_status))
984       {
985         (void)rbe_user_error(0,
986           "Internal error: Cannot get drives
                   acquired, cannot continue.");
988         return -1;
989       }

991       if(0 != GetTQDriveConcurrency(TrailID,
992                                     &DriveConcurrencyForTrail,
993                                     &temp_status))
994       {
995         (void)rbe_user_error(0,
996           "Internal error: Cannot get drive
                   concurrency, cannot continue.");
997         return -1;
998       }

1000      *QuitFlag = CancelRestoreTest();

1002      if((DriveAcquiredForTrail < DriveConcurrencyForTrail) &&
1003         ((*CountDrivesInUse) < CountDrivesAvailable) &&
1004         (FALSE == *QuitFlag))
1005      {
1006        continue;
1007      }
1008      else
1009      {
1010        break;
1011      }
1012    }
1013    return CountOfWorkItemRestoresStarted;
1014  } /* RunWorkItemRestoresForTrail() */
```

```
1018        /* Stub */
1019        static int DetermineGlobalDriveUse()
1020   1    {
1021   1        /* Limiting to MAXINT === not limiting... Need resource management
1022   1         * to do this properly.
1023   1               NOTE: This should now work like eb_dc_restore does.
1024   1         */
1025   1        return MAXINT;
           }
```

```
1028        static int
1029        SendRunningWorkItemsQuit()
1030   1    {
1031   1        int *APlist;
1032   1        int count;
1033   1        int status;
1034   1        int index;

1036   1        if(0 != getPIDList(&count, &APlist, &status))
1037   2        {
1038   2            (void)rbe_user_error(0,
1039   2                "Internal error: Cannot get auxproc pid list,
                     cannot continue.");
1040   2            return -1;
1041   1        }

1043   1        for(index = 0; index < count; index++)
1044   2        {
1045   2            QuitWorkItemRestore(APlist[index]);
1046   1        }
1047   1        return 0;
1048        }
```

```
1050        /*
1051         * Stub this out for now.
1052         */
1053        static int
1054        InterpretWorkItemRestoreResults(wi_restore_results *results)
1055  1     {
1056  1        return 0;
1057        }
```

```
1059        static void
1060        DebugLogFds(char *error_msg,
1061  1                 fd_set *fds)
1062  1     {
1063  1        int index, fd_count = 0;
1064  1        char buffer[4096];
1065  1        char *bufptr = (char*)buffer;

1068  1        for(index=0;
1069  1            index < 1024;
1070  1            index++)
1071  2        {
1072  2           if( FD_ISSET(index, fds))
1073  3           {
1074  3              int size = 0;
1075  3              size = sprintf(bufptr, "%d,", index);
1076  3              bufptr += size;
1077  3              fd_count++;
1078  2           }
1079  1        }
1080  1        rbe_log_stats(0,"%s fd_count:: %d :: {%s}\n",
1081  1                      error_msg, fd_count, buffer);
1082        }
```

```
1085      static int
1086      test_fd(int fd)
1087  1   {
1088  1       fd_set read_fd;
1089  1       int ret_select;
1090  1       struct timeval timeout = {0, 0};

1092  1       FD_ZERO(&read_fd);

1094  1       FD_SET(fd, &read_fd);

1096  1       do
1097  2       {
1098  2           ret_select = select(fd + 1, &read_fd, NULL, NULL, &timeout);
1100  1       } while((-1 == ret_select) && (EINTR == errno));

1102  1       return ret_select;

1104      }
```

```
1106      /*
1107       * test_fd_hup()
1108       *
1109       * Description: Test the supplied file descriptor to see if
1110       *   it has had the hang up condition.
1111       *
1112       * Args:
1113       *   Input fd -- the file descriptor to check for the hang up condition.
1114       *
1115       * Returns:
1116       *   1 for HUP event received on fd.
1117       *   0 No HUP event received on fd.
1118       *   -1 errno set.
1119       *
1120       */
1121      static int
1122      test_fd_hup(int fd)
1123  1   {
1124  1       struct pollfd fds;
1125  1       int ret_poll;

1127  1       if(fd < 0)
1128  2       {
1129  2           errno = EINVAL;
1130  2           return -1;
1131  1       }

1133  1       fds.fd = fd;
1134  1       fds.events = POLLIN;
1135  1       fds.revents = 0;  /* initialize */

1137  1       do
1138  2       {
1139  2           ret_poll = poll(&fds, 1, 0);
1141  1       } while((-1 == ret_poll) && (EINVAL == errno));

1143  1       if(-1 == ret_poll)
1144  2       {
1145  2           return -1;
1146  1       }

1148  1       if(POLLHUP & fds.revents)
1149  2       {
1150  2           return 1;
1151  1       }
1152  1       else
1153  2       {
1154  2           return 0;
1155  1       }

1157      } /* end test_fd_hup() */
```

```
1    /******************************************************************
2    **
3    ** File Name:   RSLauxmgr.c
4    **
5    ** Copyright (c) 1998,1999 by EMC Corporation.
6    **
7    ** Purpose:
8    ** ----------
9    **    The intent of the contents of this file is to implement the
10   **    functions the control execution of work item restores Or the
11   **    running of auxproc.
12   **
13   **       Library.
14   **
15   **    These functions are provided to allow:
16   **    - The pipe, fork, duping closing and exec of auxproc.
17   **    - The starting of work item restores.
18   **    - The quitting of work item restores.
19   **    - The getting results of work item restores.
20   **    - The getting results of work item restores.
21   **
22   **
23   **
24   **    The following functions comprise restoral management:
25   **
26   **
27   **
28   **    Compile-Time Options:
29   **
30   **       This section must list any compile time definitions
31   **       which will affect this header.
32   **
33   ******************************************************************/

36   /*
37    * Feature test switches.
38    * Standard defines required to turn on OS features go here.
39    *
40    * The following is required for code that uses POSIX API's.
41    * Remove for non-POSIX, non-portable code.
42    */

44   #define _POSIX_SOURCE 1

47   /*
48    * System headers.
49    */
51   #include <sys/wait.h>
52   #include <sys/types.h>
53   #include <unistd.h>
54   #include <string.h>
55   #include <stdlib.h>

57   /*
58    * Epoch headers.
59    */
60   #include <eb/eb_port.h>
61   #include <eb/rb_log.h>
62   #include <ebutil/eb_normalize.h>
63   #include <ebutil/ebutil.h>
64   #include <ebreport/ebvl.h>
```

```
66   /*
67    * Local headers
68    */
70   #include <RSLinterns.h>
71   #include <RSLrbrmain.h>
72   #include <restore/EDMRESubmitApi.h>
73   #include <EDMfork.h>
74   #include <RSLauxSupp.h>

76   #define SUBMIT_FIELD_MAX 2048

78   extern int putenv(const char *string);

80   extern char *strsignal(int sig);

82   extern int ChildDone(int child_pid, int *child_result);

84   static void
85   reset_recovery_privileges(struct recover_context *rcx,
86                             int changed);

88   static void
89   set_recovery_privileges(struct recover_context *rcx,
90                           int *changed);

92   static char *
93   generate_rcmdpath(int SubmitObjectID, int SubmitElemID);

95   /*
96    * StartupAuxprocess()
97    *
98    * Description:
99    *   This function pipe, fork, & exec auxproc. After which it tests
100   *   its the just started auxproc with the Ping command.  Some additional
101   *   processing is it closes the fds that are not associated with auxproc
102   *   but are inherited from the parent (restore engine). If any environment
103   *   variable need to be set for auxproc, they are set.
104   *
105   * Args:
106   *   (Inp)  debugmode -- int 1 for debug and 0 for no debug.
107   *   (Out)  struct auxproc *xp -- preallocated structure to return vital
108   *                               info for auxproc.
109   *   (char) **auxproc_envp -- environment variables to be appended to
110   *                            auxproc's environment.
111   *                            The rest of the environment is
112   *                            inherited from auxproc.
113   *                            Format is "ENV=value\0"
114   *   char *socketclientNm -- used for the client initiated restore,
115   *                           this is the client name
116   *   int clientSocketPort -- Used to identify the port number on which
117   *                           to contact the client
118   * Notes:
119   *   auxproc_envp can be NULL indicating no environment to append to
120   *   auxproc.
121   *   This is a char ** which last char * should be NULL.
```

```
 122     /*
 123      * static eerrno_ty setup_aux_processes(struct recover_context *rcx)
         */

 125 eerrno_ty
 126 StartupAuxprocess(int          debugmode,
 127     struct          auxproc *xp,
 128     char            **auxproc_envp,
 129     char            *socketClientNm,
 130     int             clientSocketPort)
 131 {

 133     #define RFD    0    /* in a pipe, fd 0 is the read descriptor */
 134     #define WFD    1    /* and fd 1 is the write descriptor */
 135     int save_errno;
 136     int fd;
 137
 138     /*
 141      * NOTES::
 142      * 1) Do I really want to be reliant on the recover_context struct.
 143      * 2) I need to fork() exec() auxproc.
 144      * 3) Is fork1 really going to work.
 145      */
 146     char *resultsbuf = NULL;
 147     char *Auxproc_pathname = AUXPROCPATHNAME;
         char *Auxproc_executable = AUXPROCNAME;

 149     int ping_status;
 150     int auxproc_index = 0; /* alton Remove */
 151     int cmd_pipe_to[2];
 152     int cmd_pipe_from[2];
 153     int bulk_pipe_to[2];
 154     int prog_pipe_from[2];

 156     if (pipe(cmd_pipe_to) == -1 ||
 157         pipe(cmd_pipe_from) == -1 ||
 158         pipe(bulk_pipe_to) == -1 ||
 159         pipe(prog_pipe_from) == -1)
 160     {
 161         save_errno = errno;
 162         rbe_log_stats(RBRECOVER_MKERR(errno), "pipe() failed");
 163         return(RBRECOVER_MKERR(save_errno));
         }

 165     /* This below appends environment variables to
 166      * the environment that auxproc inherits from the
 167      * restore engine.
 168      */
 169     if(NULL != auxproc_envp)
 171     {
 172         int index;
 173         for(index=0; NULL != auxproc_envp[index]; index++)
 174         {
 175             if(0 != putenv(auxproc_envp[index]))
 176             {
 177                 rbe_log_stats(RBRECOVER_MKERR(errno),
 178                     "Unable to set auxproc environment %s,
 179                     for auxproc PID %d.",
 180                     auxproc_envp[index], getpid());

 182                 _exit(1); /* We are the child */
 183             }
 184         }
 186     }
```

```
 188     switch (xp->xp_pid = EDMfork1())
 189     {
 190     case -1: /* Error */
 191         save_errno = errno;
 192         rbe_log_stats(RBRECOVER_MKERR(errno), "fork() failed");
 193         return(RBRECOVER_MKERR(save_errno));

 195     case 0: /* child */

 198         char procnum_str[32];
 199         char r_fd_str[32];
 200         char w_fd_str[32];
 201         char r_bulk_fd_str[32];
 202         char w_prog_fd_str[32];
 203         char dbgmodestr[32];
 204         char socket_host_str[256];                    /* Not sure what this should be */
 205
 206         char socket_port_str[32];
 207         int ret_exec = 0;
 208         socket_port_str[0] = '\0';

 210         (void)sprintf(procnum_str, "%d", auxproc_index);
 211         (void)sprintf(r_fd_str, "%d", cmd_pipe_to[RFD]);
 212         (void)sprintf(w_fd_str, "%d", cmd_pipe_from[WFD]);
 213         (void)sprintf(r_bulk_fd_str, "%d", bulk_pipe_to[RFD]);
 214         (void)sprintf(w_prog_fd_str, "%d", prog_pipe_from[WFD]);
 215         (void)sprintf(dbgmodestr, "%d", debugmode);
 216         if (NULL != socketClientNm)
 217         {
 218             (void)sprintf(socket_host_str, "%s", socketClientNm);
 219             (void)sprintf(socket_port_str, "%d", clientSocketPort);
 220         }
 221         else
 222         {
 223             socket_host_str[0] = 0;
 224             socket_port_str[0] = 0;
 226         }
 227
 228         ret_exec = execlp(
 229             Auxproc_pathname,
 230             Auxproc_executable,              /* prog to execute */
                                                 /* argv 0 */
 231             procnum_str,
 232             r_fd_str,
 233             w_fd_str,
 234             r_bulk_fd_str,
 235             w_prog_fd_str,
 236             dbgmodestr,
 237             socket_host_str,
 239             socket_port_str,
 240             (char *)0);

 241         rbe_log_stats(RBRECOVER_MKERR(errno),
 242             "Unable to exec %s, for %s PID %d.",
 243             AUXPROCNAME,
                 AUXPROCNAME,
 246             getpid());

 247         _exit (1);
         }

 249     default: /* parent */
```

```
251 2    /*
252 2     * The parent has no need for the fd
253 2     * used to write *to* the parent, nor
254 2     * the fds used to read *from* the parent.
255 2     * If these are not close EPIPE and SIGPIPE
256 2     * may be missed by the writer when the
257 2     * intended reader dies.
258 2     */

260 2    (void)close(cmd_pipe_to[RFD]);
261 2    (void)close(bulk_pipe_to[RFD]);
262 2    (void)close(cmd_pipe_from[WFD]);
263 2    (void)close(prog_pipe_from[WFD]);

265 2    /*
266 2     * but does want to save the other fds
267 2     */

269 2    xp->xp_fd_from_x    = cmd_pipe_from[RFD];
270 2    xp->xp_fd_to_x      = cmd_pipe_to[WFD];
271 2    xp->xp_fd_bulk_to_x = bulk_pipe_to[WFD];
272 2    xp->xp_fd_prog_from_x = prog_pipe_from[RFD];
273 2    /*
274 2     * In debugmode, exec the separate-process
275 2     * version of the auxprocs (ptrace issue)
276 2     */

278 2    if (debugmode)
279 3    {
280 3        auxcmdpacket(xp->xp_fd_to_x, 'X', 0, "");
281 2    }
282 1
283 1  #define PING_TEST_STR "abc"
284 1  #define PING_TEST_STR_SIZE 4 /* include the '\0' */

286 1    fd = xp->xp_fd_to_x;
287 1    auxcmdpacket(fd, 'P', PING_TEST_STR_SIZE, PING_TEST_STR);

289 1    fd = xp-> xp_fd_from_x;

291 1    ping_status = auxresults(fd, 'P', 0, &resultsbuf);

293 1    if ((-1 == ping_status) ||
294 1        (NULL == resultsbuf) ||
295 1        (strcmp(resultsbuf, PING_TEST_STR) != 0))
296 2    {
297 2        rec_api_log_csm(SUB_CSM_NO_PING_AUXPROC, NULL);
298 2        rbe_log_stats(0, "%s ping start-up",
299 2            AUXPROCNAME);
300 2        return(EP_RB_RECOVER_AUXPROC_DIED);
301 1    }

303 1    free(resultsbuf);

305 1    return(E_SUCCESS);

307 1  #undef PING_TEST_STR
308 1  #undef PING_TEST_STR_SIZE
309 1  #undef RFD
310 1  #undef WFD
311 1  #undef MAX_FD
312 1  }   /* end of setup_aux_process() */
```

```
315 1  #define MAX_SUBMIT_FIELD 2048

318    /************************************************************
319     * start_cpiogen()
320     *
321     * This function initiates a work item restore. It first determines
322     * the size of the restore command to send to auxproc, malloc's the
323     * memory and creates the restore command and sends it to auxproc.
324     * Auxproc will start the rcmd (if necessary) and start xcpiogen.
325     * Auxproc will send the initialization reply which is read by this
326     * function. The results are for initialization.
327     *
328     * Args:
329     *  (I) rcx  -- Recover Context struct (Limited use)
330     *  (I) xp  -- auxproc struct pointer.
331     *  (I) submitObjectID -- indentifies what and how to run restore.
332     *  (I) submitElemID  -- idententifies what and how to run witem
333     *            restore.
334     *  (I) auxprocnum -- auxproc number may become obsolete.
335     *  (I) *rcmdinfo[4] -- the rcmdinfo must be included.
336     *  (O) results  -- Of the work item restore initialization.
337     *  (O) err_str -- Of work item initialization failures.
338     *
339     * Returns:
340     *  xcpiogen's pid for success, -1 for failure.
341     *
342     * NOTES::
343     *  rcx -- The recover context structure is carefully used below.
344     *  The rcx structure should be used only to get the global values
345     *  like the xcpiogen executable name and the config structure.
346     *
347     * Submit Object should be used to determine the user id,
348     * admin priveledges, and other values that would not vary
349     * for a potentially multi work item restore.
350     *
351     * Submit Element should be used to determine anything that
352     * could be potentially unique for a work item restore.
353     *
354     * rcmdinfo is the command line for the remote command.
355    ************************************************************/
356  int
357  start_cpiogen(struct recover_context *rcx,
358        struct auxproc *xp,
359        int submitObjectID,
360        int submitElemID,
361        int auxprocnum,
362        char *rcmdinfo[4],
363        rcmd_pkt0_info *results,
364        char **err_str)
365  {
366    char *auxproc_databuf;
367    size_t data_len = 0;
368    char *p;
369    int i;
370    int resfd;

371    /* For Initialization results */
372    int pid = -1;
373    rcmd_pkt0_info *pkt0p;
374    rcmd_pkt0_info pkt0_buffer;
375    char *resultsbufptr = NULL;
376    char *errstr = "";
377    struct mark_summary submit_summary;
```

```
379 1    /* Args for xcpiogen */
380 1    char *xcpiogen_argv0;
381 1    int xcpiogen_argc;
382 1    char *submit_file_flag = "-S";
383 1    char *outputfd_flag = "-f%d";
384 1    char *progress_report_flag = "-p";
385 1    u_hyper total_bytes;
386 1    char *total_bytes_flag = "-B";
387 1    char *total_bytes_stringP;
388 1    char *bufsizeP = NULL;
389 1    char bufsize_buffer[20];

391 1    RBC_WORKGROUP *pg;
392 1    RBC_WORKITEM *pi;

394 1    /* temporary variable for the submit object / element fields. */
395 1    char temp_effective_uidname[MAX_SUBMIT_FIELD];
396 1    char temp_socket_host[MAX_SUBMIT_FIELD];
397 1    char temp_workitem_name[MAX_SUBMIT_FIELD];
398 1    char temp_submit_file[MAX_SUBMIT_FIELD];
399 1    int temp_socket_port;
400 1    int GetSEStatus = 0;
401 1    int GetSOStatus = 0;                         /* From the summary */

403 1    if( GetSEWorkItemName(submitObjectID, submitElemID,
404 1            temp_workitem_name, MAX_SUBMIT_FIELD,
405 1            &GetSEStatus) != 0)
406 1    {
407 2        rbe_log_stats(0, "Unable to get work item name.");
408 2        return -1;
409 1    }

411 1    if( GetSOEffectiveUserName(submitObjectID,
412 1            temp_effective_uidname,
413 1            MAX_SUBMIT_FIELD, &GetSOStatus) != 0)
414 2    {
415 2        rbe_log_stats(0, "Unable to get user name.");
416 2        return -1;
417 1    }

420 1    if( GetSERcmdConnect(submitObjectID, submitElemID,
421 1            temp_socket_host, MAX_SUBMIT_FIELD,
422 1            &temp_socket_port, &GetSEStatus) != 0)
423 2    {
424 2        rbe_log_stats(0, "Unable to get socket host/port pair.");
425 2        return -1;
426 1    }

428 1    if( GetSESubmitFile(submitObjectID, submitElemID,
429 1            temp_submit_file, MAX_SUBMIT_FIELD,
430 1            &GetSEStatus) != 0)
431 1    {
432 2        rbe_log_stats(0, "Unable to get submit file.");
433 2        return -1;
434 1    }

437 1    /*
438 1     * +1 for '\0' characters
439 1     */
441 1    data_len += strlen(rcmdinfo[0]) + 1;    /* rcmd-hostname */
442 1    data_len += strlen(rcmdinfo[1]) + 1;    /* rcmd-locuser */
443 1    data_len += strlen(rcmdinfo[2]) + 1;    /* rcmd-remuser */
```

```
444 1    data_len += strlen(rcmdinfo[3]) + 1;                         /* rcmd-cmd */

446 1    data_len += strlen(temp_effective_uidname) + 1;

448 1    data_len += sizeof (int);                                    /* fd info */
449 1    data_len += sizeof (int);                                    /* flags */
450 1    data_len += strlen(rcx -> rc_cpiogen_executable) + 1;        /* xcpiogen-cmd */
451 1    data_len += sizeof (int);                                    /* xcpiogen-argc */

453 1    /*
454 1     * xcpiogen arguments. Pass traditional argv[0]
455 1     * plus the output file descriptor number,
456 1     * plus -p (progress report mode) and -B with its
457 1     * argument (total bytes to be processed).
458 1     */
460 1    xcpiogen_argc = 7;

462 1    xcpiogen_argv0 = strchr(rcx -> rc_cpiogen_executable, '/');
463 1    if (NULL != xcpiogen_argv0)
464 2    {
465 2        ++xcpiogen_argv0;
466 1    }
467 1    else
468 2    {
469 2        xcpiogen_argv0 = rcx -> rc_cpiogen_executable;
470 1    }

472 1    data_len += strlen(xcpiogen_argv0) + 1;                 /* xcpiogen argv[0] */
473 1    data_len += strlen(temp_submit_file) + 1;               /* xcpiogen argv[1] */
474 1    data_len += strlen(outputfd_fmt) + 1;                   /* xcpiogen argv[2] */
475 1    data_len += strlen(progress_report_flag) + 1;           /* xcpiogen argv[4] */

477 1    /*
478 1     * Get the current total number of bytes to be processed from
479 1     * the mark summary u_hyper so that we can then convert it to
480 2     * a decimal string to be passed to xcpiogen().
481 1     */
483 1    if(0 != GetSEMarkedSummary(submitObjectID,
484 1            submitElemID,
485 1            &submit_summary,
486 1            &GetSEStatus))
487 2    {
488 2        /* This is not a critical error. This may cause progress
489 2         * reporting problems!
490 2         */
492 2        rbe_log_stats(0, "Unable to set submit size for xcpiogen.");
493 2        total_bytes = ul_to_uh(0);
494 1    }
495 1    else
496 2    {
497 2        total_bytes = submit_summary.len_mkd_files;
498 1    }

500 1    total_bytes_stringP = u_hyper_to_decimal(total_bytes);

502 1    /*
503 1     * Add the size of the "total bytes" flag and value string
```

```
504  1      */
506  1      data_len += strlen(total_bytes_flag) + 1;     /* xcpiogen argv[5] */
507  1      data_len += strlen(total_bytes_stringP) + 1;
                                                          /* xcpiogen argv[6] */
509  1      /*
510  1       * Add size for db API socket info
511  1       */
513  1      data_len += sizeof (int);
514  1      data_len += strlen(temp_socket_host) + 1;     /* socket port # */
516  1      /*
517  1       * locate work item in config info & get length of filespec
518  1       * %%% changed break in inner loop to 'goto' to resume with
519  1       * found item.
520  1       */
522  1      for (pi = NULL, pg = rcx->rc_config->pgrouplist /* OK */;
523  1           NULL != pg;
524  1           pg = pg->next)
525  2      {
526  2          for (pi = pg->pwlist; NULL != pi; pi = pi->next)
527  3          {
528  4              if (0 == strcmp(pi->name, temp_workitem_name))
529  4              {
530  4                  goto stopsearch;         /* %%% exit both loops */
531  3              }
532  2          }
533  1      }
535  1  stopsearch:
536  1      if (pi != NULL)
537  2      {
538  2          data_len += strlen(pi->list)+1;
539  1      }
540  2      else
541  2      {
542  2          data_len += 1;
543  1      }
545  1      {
546  2          sprintf(bufsize_buffer, "-R%d", pi->recover_server_bufsize);
547  2          bufsizeP = bufsize_buffer;
548  2          data_len += strlen(bufsizeP) + 1;
549  2          xcpiogen_argc++;                 /* one more arg to xcpiogen */
550  2      }
551  1
553  1      data_len += strlen(temp_workitem_name) + 1;
555  1      /*
556  1       * Allocate memory to hold all this gunk we need to
557  1       * shove towards our auxiliary process.
558  1       */
560  1      auxproc_databuf = sm_fmalloc((unsigned)data_len);
562  1      /*
563  1       * Fill in the gunk. First, the rcmd info for the rsh part.
564  1       */
566  1      p = auxproc_databuf;
```

```
567  1      for (i = 0; i < 4; i++)
568  2      {
569  2          (void)strcpy(p, rcmdinfo[i]);
570  2          p += strlen(p)+1;
571  1      }
573  1      /*
574  1       * The human username which the remote should operate as
575  1       */
577  1      (void)strcpy(p, temp_effective_uidname);
578  1      p += strlen(p)+1;
580  1      /*
581  1       * The xcpiogen-cmd-fd-info, which tells auxproc which
582  1       * arg has the sprintf format string to insert the
583  1       * actual output fd number. Remember auxproc sets up
584  1       * the connections between xcpiogen and the rcmd. This
585  1       * below arg is sent to auxproc to tell auxproc which
586  1       * argument to update the format sting with the outputfd.
587  1       */
589  1      i = 3;                      /* For argv[3] -- See outputfd_fmt below.*/
591  1      memcpy(p, &i, sizeof(int));
593  1      p += sizeof (int);
595  1      /*
596  1       * the flags, which are always zero currently
597  1       */
599  1      i = 0;
600  1      memcpy(p, &i, sizeof (int));
601  1      p += sizeof (int);
603  1      /*
604  1       * The "xcpiogen command" that we will run locally.
605  1       */
607  1      (void)strcpy(p, rcx -> rc_cpiogen_executable);
608  1      p += strlen(p)+1;
610  1      /*
611  1       * The argc for the "xcpiogen command", and its argv vector.
612  1       */
614  1      memcpy(p, &xcpiogen_argc, sizeof (int));
615  1      p += sizeof (int);
617  1      /*
618  1       * The argv vector, which is argv0, the outputfd
619  1       * thingy, and the progress report mode flag.
620  1       */
622  1      (void)strcpy(p, xcpiogen_argv0);
623  1      p += strlen(p)+1;
625  1      (void)strcpy(p, submit_file_flag);      /* xcpiogen argv[1] */
626  1      p += strlen(submit_file_flag) + 1;
628  1      (void)strcpy(p, temp_submit_file);      /* xcpiogen argv[2] */
629  1      p += strlen(temp_submit_file) + 1;
631  1      (void)strcpy(p, outputfd_fmt);          /* xcpiogen argv[3] */
632  1      p += strlen(outputfd_fmt) + 1;
```

```
634  1          (void)strcpy(p, progress_report_flag); /* xcpiogen argv[4] */
635  1          p += strlen(progress_report_flag) + 1;

637  1          if (bufsizeP != NULL)
638  1          {
639  2              (void)strcpy(p,bufsizeP); /* xcpiogen argv[??] */
640  2              p += strlen(bufsizeP) + 1;
641  1          }

643  1          /*
644  1           * Follow this with the total bytes flag and value.
645  1           */
647  1          strcpy(p, total_bytes_flag); /* xcpiogen argv[??] */
648  1          p += strlen(total_bytes_flag) + 1;
649  1          strcpy(p, total_bytes_stringP); /* xcpiogen argv[??] */
650  1          p += strlen(total_bytes_stringP) + 1;

652  1          /*
653  1           * socket info for db API
654  1           */
656  1          (void)memcpy(p, (char *)&temp_socket_port, sizeof (int));
657  1          p += sizeof (int);

659  1          /*
660  1           * socket host name for db API
661  1           */
663  1          (void)strcpy(p, temp_socket_host);
664  1          p += strlen(p)+1;

666  2          if (NULL != pi) /* send filespec */
667  2          {
668  2              (void)strcpy(p, pi->list);
669  2              p += strlen(p)+1;
670  1          }
671  1          else
672  2          {
673  2              *p++ = 0;
674  1          }

676  1          /*
677  1           * workitem name
678  1           */
680  1          (void)strcpy(p, temp_workitem_name);
681  1          p += strlen(p)+1;
683  1      }

684  1      /*
685  1       * assert that our arithmetic above was done correctly
687  1       */
688  2      if ((size_t)(p - auxproc_databuf) != data_len)
689  2      {
690  2          rbe_log_stats(0, "assertion failed: cmd size miscount");
691  1          return -1;
693  1      }

694  1      /*
695  1       * Send the restore command to auxproc. Auxproc will start
696  1       * The remote command (if necessary) and xcpiogen.
698  1       */
698  1      auxcmdpacket(xp-> xp_fd_to_x,
```

```
699  1                   'r', (int)data_len, auxproc_databuf);

701  1      /*
702  1       * Obtain the fork status
703  1       */
705  1      resfd = xp-> xp_fd_from_x;

707  1      i = auxresults(resfd, 'O', 0, &resultsbufptr);
709  1      if (i < 0)
710  2      {
711  2          rbe_log_stats (0,
712  2              "** Error while starting auxproc for work item \"%s\"",
713  2              temp_workitem_name);
714  2          null_free (resultsbufptr);
715  2          return -1;
716  1      }

718  1      /* This memory management is crap */
720  1      pkt0p = &pkt0p_buffer;
721  1      memcpy(pkt0p, resultsbufptr, sizeof(pkt0p_buffer));
722  1      memcpy(resultsbufptr, resultsbufptr, sizeof(pkt0p_buffer));

724  1      if ((pkt0p->msglen) > 0)
725  2      {
726  2          errstr = resultsbufptr + sizeof *pkt0p;
727  2          *err_str = esl_strdup(errstr);
728  1      }
729  1      else
730  2      {
731  2          errstr = "";
732  2          *err_str = esl_strdup("");
733  1      }

735  1      /*
736  1       * if the fork failed, the cpiogen start fails
737  1       */
739  1      if (0 != pkt0p->failcode)
740  2      {
741  2          int jnk;

744  2          if (strlen(errstr) > (size_t)0)
745  2          {
746  3              rbe_log_stats(0,
747  3                  "** Error while starting auxproc, error %s, "
748  3                  "for work item \"%s\"",
749  2                  errstr,
                       temp_workitem_name);
751  2          free (resultsbufptr);

753  2          /*
754  2           * collect the useless 'r' reply packet
755  2           */
757  2          resultsbufptr = (char *)&jnk;
758  2          (void)auxresults(resfd, 'r', sizeof (int), &resultsbufptr);
759  2          return -1;
760  1      }
```

```
762  1      /*
763  1       * Caller assumes responsibility for (eventually)
764  1       * collected exit status of remote and local programs.
765  1       */

767  1      pid = pkt0p->pid;

769  1      free (resultsbufptr);

771  1      return pid;
772  1      }    /* end of start_cpiogen() */
```

```
775  1     /*
776  1      * 1) Remove rcx references.
777         */

779     static char *
780     make_remote_cpiogen_cmd(struct recover_context *rcx,
781                             int SubmitObjectID,
782                             int SubmitElemID)
783  1     {
784  1      char *rcmdpath = generate_rcmdpath(SubmitObjectID,
785                                             SubmitElemID);
786  1      char bufsize_buffer[20];
787  1      char *bufsizeP = "";
788  1      RBC_WORKITEM *work_itemP;
789  1      unsigned len;
790  1      char *cmd;
791  1      char *remdebugflag = "";
792  1      char *noclobber = "";
793  1      char *minus_c_arg = "";
794  1      char *minus_c = "";

796  1      char temp_dirtop[SUBMIT_FIELD_MAX];
797  1      char temp_workitem_name[SUBMIT_FIELD_MAX];
798  1      OverwritePolicy temp_overwrite_policy;
799  1      int GetSEStatus = 0;

801  1      if (NULL == rcmdpath)
802  2      {
803  2          return NULL;
804  2      }

805  1      temp_overwrite_policy = GetSEDestOverWritePolicy(SubmitObjectID,
806  1                                                       SubmitElemID,
807  1                                                       &GetSEStatus);

809  1      if (0 != GetSEStatus)
810  2      {
811  2          rbe_log_stats(0, "Unable to get overwrite policy.");
812  1          return NULL;
813  1      }

815  1      if (GetSEDestDirTop(SubmitObjectID, SubmitElemID,
816  1                          temp_dirtop, SUBMIT_FIELD_MAX,
817  1                          &GetSEStatus) != 0)
818  1      {
819  2          rbe_log_stats(0, "Unable to get dirtop.");
820  2          return NULL;
821  1      }

823  1      if (GetSEWorkItemName(SubmitObjectID, SubmitElemID,
824  1                            temp_workitem_name,
825  1                            SUBMIT_FIELD_MAX,
826  1                            &GetSEStatus) != 0)
827  2      {
828  2          rbe_log_stats(0, "Unable to get work item name.");
829  1          return NULL;
830  1      }

832  1      if (temp_dirtop != NULL)
833  2      {
834  2          minus_c = " -c ";
835  2          minus_c_arg = temp_dirtop;
836  1      }

838  1      work_itemP = rbc_find_workitem_in_config(temp_workitem_name,
```

```c
                NULL, NULL,
                rcx->rc_config, /* OK */
            RBC_FIND_WORKITEM_NO_OPTIONS);

    if (work_itemP != NULL
        && DEFAULT_EB_BUFSIZE != work_itemP->recover_client_bufsize)
    {
        sprintf(bufsize_buffer, " -A -b -A %d",
            work_itemP->recover_client_bufsize);
        bufsizeP = bufsize_buffer;
    }

    switch (temp_overwrite_policy)
    {
    case RC_OVERPOL_NO_CLOBBER:
        noclobber = " -A -onever";
        break;

    case RC_OVERPOL_NEW_CLOBBER:
        noclobber = " -A -onewer";
        break;

    default:
        break;    /* do something better here */
    }

    if (debugmode)
    {
        static char debugarg[100];

        (void)sprintf(debugarg, " -x    /tmp/RBRdebug%d    ", getpid(
        ));
        remdebugflag = debugarg;
    }

    len = strlen(rcmdpath) +
        strlen(bufsizeP) +
        strlen(remdebugflag) +
        strlen(minus_c) +
        strlen(minus_c_arg) +
        strlen(noclobber) +
        1;    /* for '\0' */

    cmd = sm_fmalloc(len);
    (void)sprintf(cmd, "%s%s%s%s%s%s",
        rcmdpath,
        bufsizeP,
        remdebugflag,
        minus_c,
        minus_c_arg,
        noclobber);

    return cmd;
    /* end of make_remote_cpiogen_cmd() */
}
```

```c
/*
 * 1) Remove rcx references.
 * 2) Research whether ebc_normalize updates.
 */
/*
 * Construct the path name of the remote command that
 * will be executed on the destination client.
 *
 * Return ptr to constructed string.
 * NOTE: caller must copy if string is to be preserved.
 */
static char *
generate_rcmdpath(int SubmitObjectID,
                  int SubmitElemID)
{
    static char *mybuf = NULL;
    size_t len_needed;
    char *pph;    /* pointer to %h */
    char *p;
    char *q;
    char *norm_host;

    char temp_scriptname[SUBMIT_FIELD_MAX];
    char temp_client_hostname[SUBMIT_FIELD_MAX];

    int GetSEStatus = 0;

    if ( GetSERcmdScriptName(SubmitObjectID, SubmitElemID,
            temp_scriptname, SUBMIT_FIELD_MAX,
            &GetSEStatus) != 0)
    {
        rbe_log_stats(0, "Unable to get rcmd script name.");
        return NULL;
    }

    if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
            temp_client_hostname, SUBMIT_FIELD_MAX,
            &GetSEStatus) != 0)
    {
        rbe_log_stats(0, "Unable to get client destination name.");
        return NULL;
    }

    /*
     * If there is a %h in the rc_client_scriptname,
     * that means put the hostname in there.
     *
     * Sorry, no escapes implemented.  You can't have an
     * rc_client_scriptname with a literal %h in it.  Tough.
     */
    for (pph = temp_scriptname; *pph != '\0'; pph++)
    {
        if (*pph == '%' && *(pph+1) == 'h')
            break;
    }

    /*
     * no %h, just use the bare client scriptname
     */
    if (*pph == '\0')
```

```
 959  2          return temp_scriptname;
 960  1      }
 961  1
 962  1      /*
 963  1       * there is a %h ... insert the destination client
 964  1       * name into the rc_client.scriptname.  First
 965  1       * compute how much storage will be needed to
 966  1       * hold the result.
 967  1       */
 969  1      if (!ebc_can_it_be_normalized(temp_client_hostname))
 970  2      {
 971  2          rbe_log_stats(
 972  2              rec_api_log_csm(SUB_CSM_NOMEM, NULL);
 973  2              0, "Could not allocate memory generate_rcmdpath");
 974  2          /*
 975  2          StartDoneCallBack(EP_RB_RECOVER_FATALERR);*/
 976  2          return NULL;
 977  1      }
 978  1      norm_host = ebc_normalize(temp_client_hostname);
 979  1      len_needed =
 980  1          strlen(temp_scriptname) - 2 +          /* -2: %h */
 981  1          strlen(norm_host) + 1;                 /* +1: '\0' */
 982  1      if (mybuf == NULL || strlen(mybuf) < (size_t)(len_needed-1))
 983  2      {
 984  2          if (mybuf != NULL)
 985  3          {
 986  3              free(mybuf);
 987  2          }
 989  2          if ((mybuf = malloc((int)len_needed)) == NULL)
 990  2          {
 991  3              rbe_log_stats(
 992  3                  rec_api_log_csm(SUB_CSM_NOMEM, NULL);
 993  3                  0, "Could not allocate memory generate_rcmdpath");
 994  2              return NULL;
 995  2          }
 996  1      }
 997  1      q = mybuf;
 998  1      for (q = mybuf, p = temp_scriptname; p < pph; q++, p++)
 999  2      {
1000  2          *q = *p;
1001  1      }
1003  1      (void)strcpy(q, norm_host);
1004  1      (void)strcat(mybuf, pph+2);
1006  1      return mybuf;
1007  1  }   /* end of generate_rcmdpath() */
```

```
1009  1  int
1010  1  StartWorkItemRestore(struct recover_context *rcx,
1011  1                       struct auxproc *xp,
1012  1                       int SubmitObjectID,
1013  1                       int SubmitElemID)
1014  1  {
1015  1      char *rcmdv[4];        /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
1016  1      int  changed_priv = 0;
1017  1      int  xpclogen_pid;
1018  1      long temp_effective_uid;
1019  1      rcmd_pkt0_info pkt0p_buffer;
1021  1      char *err_str_buffer = NULL;
1022  1      char temp_client_hostname[SUBMIT_FIELD_MAX];
1023  1      char temp_client_rbuname[SUBMIT_FIELD_MAX];
1024  1      boolean_ty temp_src_sysadmin;
1026  1      int GetSEStatus = 0;
1027  1      int GetSOStatus = 0;
1030  1      if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
1031  1                  temp_client_hostname,
1032  1                  SUBMIT_FIELD_MAX,
1033  1                  &GetSEStatus) != 0)
1034  2      {
1035  2          rbe_log_stats(0, "Unable to get client destination name.");
1036  2          return -1;
1037  1      }
1041  1      if ( GetSEClientUserName(SubmitObjectID, SubmitElemID,
1042  1                  temp_client_rbuname,
1043  1                  SUBMIT_FIELD_MAX,
1044  1                  &GetSEStatus) != 0)
1045  2      {
1046  2          rbe_log_stats(0, "Unable to get client user name.");
1047  2          return -1;
1048  1      }
1052  1      temp_src_sysadmin = GetSOSourceSystemAdmin(SubmitObjectID,
1053  1                  &GetSOStatus);
1054  1      if (0 != GetSOStatus)
1055  2      {
1056  2          rbe_log_stats(
1057  2              0, "Unable to get the source system admin priveledges.");
1058  2          return -1;
         1      }
1060  1      if ( GetSOEffectiveUID(SubmitObjectID,
1061  1                  &temp_effective_uid,
1062  2                  &GetSOStatus) != 0)
1063  2      {
1064  2          rbe_log_stats(0, "Unable to get effective uid.");
1065  2          return -1;
1066  1      }
1069  1      if (0 != (temp_src_sysadmin) || 0 != temp_effective_uid)
1070  2      {
1071  2          set_recovery_privileges(rcx, &changed_priv);
1072  1      }
```

```
1074 1      rcmdv[0] = temp_client_hostname;
1075 1      rcmdv[1] = temp_client_rbname;
1076 1      rcmdv[2] = temp_client_rbuname;
1077 1      rcmdv[3] = make_remote_cpiogen_cmd(rcx,
1078 1                     SubmitObjectID,
1079 1                     SubmitElemID);

1081 1      if(NULL == rcmdv[3])
1082 2      {
1083 2          return -1;
1084 1      }

1085 1      xcpiogen_pid = start_cpiogen(rcx, xp,
1086 1                     SubmitObjectID,
1087 1                     SubmitElemID,
1088 1                     0,
1089 1                     rcmdv,
1090 1                     &pkt0p_buffer,
1091 1                     &err_str_buffer);

1093 1      if (changed_priv)
1094 2      {
1095 2          reset_recovery_privileges(rcx, changed_priv);
1096 1      }

1097 1      return xcpiogen_pid;
1098 1  } /* StartWorkItemRestore() */
```

```
1100 1  /*
1101 1   * Check to see if the user has root access to the
1102 1   * destination client. If so, give him/her the root
1103 1   * privileges on the recovery.
1104 1   */
1106 1  static void
1107     set_recovery_privileges(struct recover_context *rcx,
1108                    int *changed)
1109 1  {
1110 1      int root_access;
1111 1      eperrno errnum;

1114 1      *changed = 0;
1115 1      (void)rbc_canirecover(rcx->rc_config, rcx->rc_client_hostname,
1116 1                     rcx->rc_human_uidname, &root_access,
                        &errnum);

1117 1      if (root_access)
1118 2      {
1119 2          if (debugmode)
1120 3          {
1121 3              rbe_log_stats(
1122 2                  0, "the user is a sys admin for the dest client");
1124 2          }

1125 2          rcx->rc_recovery_flags |= RC_RECFLAG_DEST_SYSADMIN;
1126 3          if (rcx->rc_effective_uid != 0)
1127 3          {
1129 4              rcx->rc_effective_uid = 0;
1130 3              rcx->rc_effective_uidname = "root";
1132 3              *changed = SET_ROOT;

1133 3              if (debugmode)
1134 3              {
1135 2                  rbe_log_stats(0, "changing the uid to admin status");
1136 1              }
1137 1          }
1138 2      }
                else
                {
1139 2          if (debugmode)
1140 3          {
1141 3              rbe_log_stats(
1142 2                  0, "the user don't have sys admin status on dest client");
1143 2              rbe_log_stats(
                        0, "effect uid = %d", rcx->rc_effective_uid);
1145 2          }

1146 2          /*
1147 2           * The user does not have root access to the dest
1148 2           * client. But s/he is the system admin for the
1149 2           * source client, therefore, we need to stripe the
1150 2           * root stuff from the user during the recovery.
                 */
1152 2          rcx->rc_recovery_flags &= ~RC_RECFLAG_DEST_SYSADMIN;
1153 2          if (rcx->rc_recovery_flags & RC_RECFLAG_SOURCE_SYSADMIN)
1154 3          {
1155 3              if (debugmode)
1156 4              {
1157 4                  rbe_log_stats(
1158 3                      0, "changing the uid to regular user status");
                    }
                }
            }
```

```
1160 3        rcx->rc_effective_uid = rcx->rc_human_uid;
1161 3        rcx->rc_effective_uidname = rcx->rc_human_uidname;
1162 3        *changed = SET_USER;
1163 2      }
1164 1    }
1165    } /* end of set_recovery_privileges() */
```

```
1168    /*
1169     * Reset the user's identity. The user may have root access
1170     * on the destination client, but not on the source, and vice
1171     * versa.
1172     */
1174    static void
1175    reset_recovery_privileges(struct recover_context *rcx,
1176                              int changed)
1177 1  {
1179 2    if (changed == SET_ROOT)
1180 2    {
1181 3      if (debugmode)
1182 3      {
1183 2        rbe_log_stats(
                  0, "resetting the uid to regular user status");
             }
1185 2      rcx->rc_effective_uid = rcx->rc_human_uid;
1186 2      rcx->rc_effective_uidname = rcx->rc_human_uidname;
           }
1187 1    else
1188 1    {
1189 2      if (debugmode)
1190 2      {
1191 3        rbe_log_stats(0, "resetting the uid to sys admin status");
1192 3      }
1193 2      rcx->rc_effective_uid = 0;
1195 2      rcx->rc_effective_uidname = "root";
1196 2    }
1197 1  } /* end of reset_recovery_privileges() */
1198
```

```
1202  1    /**
1203  1    **
1204  1    **
1205  1    **    FUNCTION DESCRIPTION:
1206  1    **
1207  1    **        This function will start the workitem restore termination.
1208  1    **
1209  1    **
1210  1    **    INPUTS:
1211  1    **        int auxproc_pid -- auxproc's pid.
1212  1    **
1213  1    **
1214  1    **    RETURN VALUE:
1215  1    **        none
1216  1    **
1217  1    **
1218  1    **    SIDE EFFECTS:
1219  1    **        auxproc sent the USR1 signal, auxproc will send xcpiogen the
1220  1    **        TERM signal to quit the restore. Auxproc will wait until the
1221  1    **        restore terminates by waiting for the remote command's exit
1222  1    **        status.
1223  1    **
1224  1    **   ++
1225  1    **
1227       **/
1228
1229  1    void
           QuitWorkItemRestore(int auxproc_pid)
           {

1232  1        /*
1233  1         * Auxproc will now alert xcpiogen by sending the
1234  1         * This will give xcpiogen the ability to
1235  1         * clean up tmpfiles and clean-up sockets.
1236  1         * xcpiogen will commit suicide as a result of
1237  1         * this signal.
1238  1         */

1241  1        /*
1242  1         * Alert the auxproc that we want out.
1243  1         * recxcpio etc should also die as a result
1244  1         * of xx_read_or_die_xx() in recx. The SIGUSR1
1245  1         * should not kill the auxproc itself, but only
1246  1         * notify auxproc of the diminishing restore.
1247  1         */

1249  1        (void)kill(auxproc_pid, SIGUSR1);

1251  1        if (debugmode)
1252  2        {
1253  2            rbe_log_stats(0, "%s %d quiting restore in process",
1254  2                AUXPROCNAME,
1255  2                auxproc_pid);
1256  1        }

1258  1        /*
1259  1         * now that we have indirectly killed the xcpiogen and alerted the
1260  1         * auxproc,
1261  1         * the signal from the auxproc will be picked up by the next
1262  1         * routine, auxprocsig_handler, it will notify the user of the
               * results and does the cleaning up.
```

```
1263  1         */
1264  1        return; /* QuitWorkItemRestore() */
1265           }
```

```
1267  /**
1268  **
1269  **   GetAuxprocResults()
1270  **
1271  **   FUNCTION DESCRIPTION:
1272  **
1273  **   Inherited from auxprocsig_handler().
1274  **
1275  **   This routine is called when there is information appears
1276  **   in the aux-process. The aux-process starts xcpiogen and
1277  **   is responsible for "listening" to status coming from xcpiogen.
1278  **   When status comes back from xcpiogen, the aux-process signals
1279  **   this process, which is trapped by the caller. And finally,
1280  **   this routine is called.
1281  **
1282  **   Args:
1283  **     (I) resfd int -- results file decriptor from auxproc.
1284  **     (O) results -- work item results.
1285  **
1286  **   RETURN VALUE:
1287  **     The number of local and remote status collected from fd.
1288  **
1289  **   SIDE EFFECTS:
1290  **     none
1291  **
1292  **/

1295 1   int
1296 1   GetAuxprocResults(int resfd,
1297 1                     wi_restore_results *results)
1298 1   {
1299 1       int      exit_stat;
1300 1       char     *resbuf;
1301 1       int      n_read;
1302 1       char     c = 0;
1303 1       int      n_status_read = 0;

1305 1       while ((n_status_read < 2) && (1 == fd_avail_test(resfd)))
1306 2       {
1307 2           resbuf = (char *) &exit_stat;

1309 2           if(1 != pread_or_warn(resfd, &c, 1, auxproc_comm_warning))
1310 3           {
1311 3               return 0;
1312 2           }

1314 2           if (debugmode)
1315 3               rbe_log_stats(
1316 3                   0, "GetAuxprocResults() called: '%c'", c);
1317 2

1319 2           if (c == 'R')
1320 3           {
1321 3               if (1 == fd_avail_test(resfd))
1322 3               {
1323 4                   /*
1324 4                    * the 'R' command is the for the remote process status
1325 4                    */
1326 4                   n_read = auxres2(resfd, 'R', sizeof(int), &resbuf);
1327 4                   if (-1 != n_read)
1328 5                   {
1329 5                       results -> remote_exit_status = exit_stat;
1330 5                       results -> remote_exit_set = TRUE;
1331 5                       n_status_read++;
```

```
1332 5                   }
1333 6                   if (debugmode)
1334 6                       rbe_log_stats(
1335 5                           0, "remote exit status obtained: %d", exit_stat);
1336 4               }
1337 4               else
1338 5               {
1339 5                   rbe_log_stats(
1340 5                       0, "Internal error: remote exit status Incomplete.");
1341 4                   return -1;
1342 3               }
1343 3           }
1344 2           else if (c == 'r')
1345 3           {
1346 3               /*
1347 3                * the 'r' command is the for the local process status
1348 4                */
1349 4               if (1 == fd_avail_test(resfd))
1350 4               {
1352 4                   n_read = auxres2(resfd, 'r', sizeof(int), &resbuf);
1353 4                   if (-1 != n_read )
1354 4                   {
1355 5                       results -> local_exit_status = exit_stat;
1356 5                       results -> local_exit_set = TRUE;
1357 5                       n_status_read++;
1358 5                   }
1359 6                   if (debugmode)
1360 6                       rbe_log_stats(
1361 5                           0, "local exit status obtained: %d", exit_stat);
1362 4               }
1363 4               else
1364 5               {
1365 5                   rbe_log_stats(
1366 5                       0, "Internal error: local exit status Incomplete.");
1367 4                   return -1;
1368 3               }
1369 2           } /* while() */
1370 2       } /* sleep (1);*/

1371 1       return n_status_read;
1373 1       /*return n_status_read;*/
1374 1   }

1376     /* GetAuxprocResults() */
```

```
1379   2   /*
1380   2    * KillWorkItemRestore()
1381   2    *
1382   2    * Kill the work item restore. Keep in mind this
1383   2    * may only be done if the work item is not running
1384   2    * a restore.
1385   2    *
1386   2    * This routine also does the waitpid for auxproc.
1387   2    * The waitpid (ChildDone()) eliminates the defunct auxproc
1388   2    * process.
1389   2    *
1390   2    * If the work item is running then one must do the
1391   2    * following.
1392   3    *
1393   3    * 1) Call QuitWorkItemRestore()
1394   3    * 2) Wait for and read results from the cmd_from pipe.
1395   3    * 3) Call KillWorkItemRestore()
1396   3    *
1397   3    * Returns:
1398   3    *   int -- Zero for success.
1399   2    *
1400   2    * Args:
1401   2    *   ap_pid -- auxproc pid.
1402   2    *   cmd_to -- auxproc cmd pipe.
1403   2    */
1404   1   int
1405   1   KillWorkItemRestore(int ap_pid, int cmd_to)
1406   1   {
1407   1       int killRet;
1408   1       int apResult;
1409   1       char *apName=AUXPROCNAME;
1410   1       char *databuf = NULL;
1411   1       int ChildDoneRet;

1413   1       killRet = kill(ap_pid, SIGTERM);

1415   1       if (-1 == killRet)
1416   2       {
1417   2           rbe_log_stats(
1418   2               0, "Can't send sigterm to \"%s\": Pid: %d, error = %s\n",
1419   2               apName, ap_pid, strerror(errno));

1420   2           return -1;
1422   2       }

1424   1       do
1425   2       {
1426   2           ChildDoneRet = ChildDone(ap_pid, &apResult);
1427   2           if(0 == ChildDoneRet) sleep (1);
1429   2       }
1430   1       while(0 == ChildDoneRet);

1431   1       switch (ChildDoneRet)
1432   2       {
1433   2           /*
1434   2            * -1  internal error. errno is set.
1435   2            *  0  child still running.
1436   2            *  1  child exited.
1437   2            *  2  child signalled (no core)
1438   2            *  3  child signalled core file generated.
1439   2            *  4  child stopped.
1440   2            */
1441   2           case(-1):
                        return -1;
                        /* no break necessary */
```

```
1443   2           case(1):
1445   1               rbe_log_stats(0,
1446   1                   "Sigterm did not bring down \"%s\": Pid: %d,"
1447   1                   " it instead exitted with = %d\n",
1448   1                   apName, ap_pid, apResult);
1449   1               break;

1451   2           case(2):
1452   2           case(3):
1453   2               if (SIGTERM != apResult)
1454   3               {
1455   3                   rbe_log_stats(0,
1456   3                       "Sigterm did not bring down \"%s\": Pid: %d, "
1457   3                       " instead killed by signal = %s\n",
1458   3                       apName, ap_pid,
1459   3                       strsignal(apResult));
1460   3               }
1461   2               break;

1463   2           case(4):
1464   2               rbe_log_stats(0,
1465   2                   "Sigterm did not bring down \"%s\": Pid: %d,"
1466   2                   " instead stopped by signal = %s\n",
1467   2                   apName, ap_pid,
1468   2                   strsignal(apResult));
1470   2               break;

1472   1       }

1475   1       /* auxcmdpacket(cmd_to, 'q', 0, databuf);*/
1476   1       return 0;
1478   1   } /* KillWorkItemRestore() */
```

```c
1    /*
2     *          RSLauxmain.c
3     *
4     * Copyright (c) 1998, 1999 EMC Corporation.  All rights reserved.
5     *
6     *
7     *
8     * -----------------------------
9     * Table of Contents:
10    * -----------------------------
11    ** static char *rb_getmethod(host, int *concurrency, int *client_type)
12    ** int looprw(int fd, char *buf, int nbytes, int (*func)());
      ** int fd_avail_1_wait_intr(int fd);
      */
14    #define  _POSIX_SOURCE 1
15    #define  E_GRANDFATHER 1
16    #define  TIME_STRUCT
17    #include <eb/eb_port.h>

19    #include <netdb.h>
20    #include <pwd.h>
21    #include <ctype.h>
22    #include <sys/wait.h>
23    #include <util/esl_select.h>
24    #include <eerrno/esl_strerror.h>
25    #include <util/esl_limit.h>
26    #include <stdlib.h>

28    /* SFP header */
29    #include <cdl/cdl.h>
30    #include <cdl/cdl_server.h>

32    #include <ebconfig/rbconfig.h>
33    #include <epffdb/ffdb.h>
34    #include <epffdb/rbfinclient.h>
35    #include <ebutil/eb_normalize.h>
36    #include <ebutil/ebsock_if.h>
37    #include <ebutil/ebutil.h>
38    #include <eb/rb_log.h>
39    #include <restore/REprogmsg.h>

41    #include <RSLinterns.h>
42    #include <RSLspexits.h>
43    #include <RSLremfd.h>
44    #include "RSLauxSupp.h"

46    /* EDMLINK API */
47    #include <edmlink/edmlink_api.h>

50    /* lint ugliness */
51    #ifndef Printf
52    #define Printf   (void)printf
53    #endif
54    #define MAX_FD 1024
55    int debugmode = 0;
56    int is_symmpath = 0;
57    int sp_cdlexitdone = 0;

59    static int xcpiogen_pid = -1;
60    static int xcpiogen_prog_fd = -1;
61    int logging_channel = -1;

63    int xcpiogen_pipe[2] = {-1, -1};
64    static char wififiledir[] = "/usr/epoch/etc/";

66    static char *rb_getmethod(
```

```c
67        register char *host, uint_t *concurrency, uint_t *client_type);
68    static void sigterm_handler(int sigvalue);   /* "attention" signal */
69    static void sigusr1_handler(int sigvalue);   /* sigterm signal */
70    static int write_CDL_no_eintr(int fd, char *buf, int nbytes);

72    static int read_CDL_no_eintr(int fd, char *buf, int nbytes);
73    static int ForwardXcpiogenProgress(int xcpiogen_prog_fd,
74                                       int restore_engine_prog_fd,
                                         boolean_ty *zero_byte_read);

76    static int DemuxAuxChildren(int progress_fd,
77                                int remote_fd,
78                                char *remote_progname,
79                                int xcpiogen_fd,
80                                int xcpiogen_pid,
81                                int *remote_exit);

83    extern int fd_avail_test(int);  /* Found in RSLauxSupp.c */

85    struct auxproc_context
      {
86        int       ap_my_auxnum;
87        int       ap_r_fd;
88        int       ap_w_fd;
89        int       ap_r_bulk_fd;
90        int       ap_w_prog_fd;
91        char      ap_cmd;
92        int       ap_datalen;
93        char      *ap_data;
94        int       ap_resultlen;
95        char      *ap_resultdata;
96        ushort_t  ap_shelltcp_port;
97        int       ap_have_shelltcp_port;
98        int       ap_socketport;
99        char      *ap_sockethost;
100       char      *ap_workitem;
101       struct    rbc_configs *ap_config;

103       char      ap_error_message[4096];
104   };
105   /*
106    * Read remote stderr output from fd.
107    * Parse it; act on it according to protocol.
108    * Remote exit status is the last thing that comes
109    * back via the remote stderr stream; return the remote
110    * exit status via *exitp.
111    */

113   enum input_states
      {
114       INSTATE_NG,
115       INSTATE_SEARCH_PREFIX0,
116       INSTATE_SEARCH_PREFIX1,
117       INSTATE_SEARCH_PREFIXN,
118       INSTATE_GATHER_COOKIE,
119       INSTATE_SEARCH_SUFFIXN,
120       INSTATE_GATHER_STATUS,
121       INSTATE_COPY_TO_STDOUT,
122       INSTATE_NEWLINE
123   };
124   static boolean_ty parse_remote_stderr_info2(int prog_fd,
125                                               int remote_fd,
126                                               int *exitp,
127                                               char *remhostname,
128                                               boolean_ty first_call,
129                                               enum input_states
                                                  *state_ptr,
```

```
130      enum input_states
131          *next_state_ptr,
132      boolean_ty
133          *skipping_leading_whitespace,
             int *parsePos,
             int *msgPos);



136  static char *recover_size_prefix(struct auxproc_context *cxp);
137  static int *ebr_direct_rcmd(char **ahost, unsigned short inport,
138          struct auxproc_context *cxp,
139          char *locuser,
             char *remuser, char *cmd, int *fd2p);
```

```
141  /*
142   * The auxiliary process(es) communicate with the main
143   * process via a simple protocol run on a pair of pipes.
144   *
145   * The parent writes commands to the auxiliary process.
146   * The format of a command is:
147   *
148   *    <cmd><data-len><data>
149   *
150   * where
151   *    <cmd> is a one-byte command
152   *    <data-len> is an "int",
                      and indicates the number of <data> bytes
153   *    <data> is command-specific data.
154   *
155   * <data-len> may be zero, but must always be present. Therefore, the
156   * minimum command "packet" is five bytes long.
157   *
158   * Result packets are written back to the parent.
159   *                    The packet format is
      * the same as the command format, though (obviously) the format
160   * of the data in a results packet is usually different from
161   * the format in a command packet.
162   *
163   * Communications are assumed to be error-free.   In other
164   * words, pipes are assumed to work correctly.
165   */

167  static int attn_ec;                /* count of SIGUSR1s (ATTN signals) */

      /*
171   * 'z' prefixes identify top-level routines
172   *    called from the auxproc switch.
173   *
174   *    There is no significant to the 'z' letter -- I just picked
175   *    a letter at random (hah!) to identify the top level funcs.
176   *
177   */

179  static void z_rcmdfilter(struct auxproc_context *cxp);
180  static void z_exec_separate_auxproc(struct auxproc_context *cxp,
181                      char *pathname);
```

```
183      int main(int argc, char *argv[])
184 1    {
185 1      char *auxproc_intial_delay_str = NULL;
186 1      int auxproc_intial_delay_int = 0;

188 1      if (argc != 9)
189 1      {
190 2        exit(1);
191 1      }

193 1      auxproc_intial_delay_str = getenv("AUXPROC_INITIAL_DELAY");
194 1      if (NULL != auxproc_intial_delay_str)
195 2      {
196 2        auxproc_intial_delay_int = atoi(auxproc_intial_delay_str);
197 2        sleep(auxproc_intial_delay_int);
198 2      }

200 1      debugmode = atoi(argv[6]);

202 1      do_auxproc(atoi(argv[1]),    /* auxproc ordinate. */
203 1                 atoi(argv[2]),    /* cmd_to auxproc pipe*/
204 1                 atoi(argv[3]),    /* cmd_from auxproc pipe*/
205 1                 atoi(argv[4]),    /* bulk_to auxproc pipe*/
206 1                 atoi(argv[5]),    /* prog_from auxproc pipe*/
207 1                 argv[0],          /* progname */
208 1                 argv[7],          /* socket host name */
209 1                 atoi(argv[8]));   /* socket port */

211 1      return(0); /* Can we get here ?? */

213      } /* End main() */
```

```
215      /*
216       * Worker-bee loop.
217       *
218       * Read instructions on r_fd.
219       * Write results to w_fd.
220       */

222      int ncmds_rcvd = 0;

224      void
225      do_auxproc(int procnum,
226                 int r_fd,
227                 int w_fd,
228                 int r_bulk_fd,
229                 int w_prog_fd,
230                 char *Xname,
231                 char *sockethost,
232                 int socketport)
233 1    {
234 1      struct auxproc_context ctx;
235 1      struct servent *sp;
236 1      sigset_t empty_set;
237 1      int rotation_size;
238 1      int errnum;
239 1      int index_fd;

241 1      /*
242 1       * close all file descriptors that we do not need
243 1       * we only need the ones passed to us from the restore
244 1       * engine. So we can close anything above stderr and anything
245 1       * that is no equal to what was passed in
246 1       */
247 1      for(index_fd = 3; MAX_FD > index_fd; index_fd++)
248 2      {
250 2        if ((index_fd == r_fd) ||
251 2            (index_fd == w_fd) ||
252 2            (index_fd == r_bulk_fd) ||
253 2            (index_fd == w_prog_fd))
254 3        {
255 3          continue;
256 2        }
257 2        (void)close(index_fd);
259 1      }

261 1      /*
262 1       * Prepare sigaction parameters
263 1       */
264 1      memset(&ctx, 0, sizeof(struct auxproc_context));

266 1      sigemptyset(&empty_set);

268 1      /*
269 1       * ignore mild keyboard interrupts (^C); parent handles
270 1       */
272 1      eb_set_signal_handler(SIGINT, SIG_IGN, &empty_set, E_SA_RESTART);

274 1      /*
275 1       * SIGUSR1 used to get our attention when the parent
276 1       * process really wants us to stop what we are doing.
277 1       */
279 1      eb_set_signal_handler(
```

```
280 1  eb_set_signal_handler(
           SIGUSR1, sigusr1_handler, &empty_set, E_SA_RESTART);
           SIGTERM, sigterm_handler, &empty_set, E_SA_RESTART);

282 1  /*
283 1   * Arguments passed are collected together in
284 1   * this context structure only because that makes
285 1   * it easier to add/change arguments in the future
286 1   * (diddle the structure rather than diddle a zillion
287 1   * calls in the switch statement).
288 1   */

290 1  ctx.ap_my_auxnum    = procnum;
291 1  ctx.ap_r_fd         = r_fd;
292 1  ctx.ap_w_fd         = w_fd;
293 1  ctx.ap_r_bulk_fd    = r_bulk_fd;
294 1  ctx.ap_w_prog_fd    = w_prog_fd;
295 1  ctx.ap_sockethost   = sockethost;
296 1  ctx.ap_socketport   = socketport;

298 1  /*
299 1   * Will need the shell/tcp service later; if
300 1   * can't find it now we might as well complain now.
301 1   */

303 1  ctx.ap_have_shelltcp_port = 0;
304 1  if ((sp = getservbyname("shell", "tcp")) != NULL)
305 2  {
306 2      ctx.ap_shelltcp_port = (ushort_t)sp->s_port;
307 2      ctx.ap_have_shelltcp_port = 1;
308 1  }
309 1  else
310 2  {
312 2      WriteFmtStringMsg(
               ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
               "Error: Cannot find shell/tcp network
                                          service.\n"
313 2          "Browsing of catalogs may continue, but"
               "the \"start\" command will not work.");
316 1  }

318 1  if (NULL == ctx.ap_config)
319 2  {
320 2      if(NULL == (ctx.ap_config = malloc(sizeof(
321 3          struct rbc_configs))))
322 3      {
323 3          /* We would prefer to log this stuff, but we
324 3           * have not opened logging yet.
325 3           */
326 3          WriteFmtStringMsg(
                   ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                   "Could not allocate memory in
                                              do_auxproc");
327 3          exit(1);
328 2      }
329 2      if (rbc_parse_config(
               NULL /*use the default name */, &ctx.ap_config,
               RBC_PARSE_DO_NOT_PRESERVE |
                                    RBC_PARSE_APPLY) != 0)
330 2      {
332 3          /* We would prefer to log this stuff, but we
333 3           * have not opened logging yet.
334 3           */
335 3  
```

```
337 3          rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
338 3          WriteFmtStringMsg(
339 3              ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                   "Auxproc -- Cannot parse configuration
                                                 file");
340 3          exit(1);
341 2      }
342 1  }

344 1  if ((errnum = eb_path_init()) != 0)
345 2  {
346 2      exit(1);
347 1  }

349 1  (void)rbe_log_init("auxproc");

351 1  /*
352 1   * Determine the "recoveries.log" file rotation size.
353 1   * Use a default unless this was specified in the config file.
354 1   */

356 1  rotation_size = RBRECOVER_LOGSIZE;
357 1  if ((ctx.ap_config != (struct rbc_configs *)NULL)
358 1      && (ctx.ap_config->srlfile.filename != NULL))
359 2  {
360 2      rotation_size = ctx.ap_config->srlfile.rotation_size;
361 1  }

363 2  (void)rbe_log_add(STATS_LOGGING, eb_recover_logpath, LOGT_FILE,
364 1      rotation_size, &logging_channel);

367 1  /*
368 1   * The actual worker-bee loop
369 1   */

371 1  for (;;)
372 2  {
373 2      char c;
374 2      int datalen;
375 2  #define SMALL_DATALEN 128
376 2      char buf[SMALL_DATALEN];
377 2      char *data;

379 2      /*
380 2       * read command byte
381 2       */
383 2      pread_or_die(r_fd, &c, 1, _exit);

385 2      /*
386 2       * read data-len
387 2       */
389 2      pread_or_die(r_fd, (char *)&datalen, sizeof datalen, _exit);

391 2      /*
392 2       * We do the read here for simple commands that do not
393 2       * take much data.
394 2       */
396 2      if (datalen > SMALL_DATALEN || datalen == 0)
397 3      {
398 2          data = NULL;
399 2      }
400 2      else
```

```
401 3    {
402 3        pread_or_die(r_fd, buf, datalen, _exit);
403 3        data = buf;
404 2    }

406 2    ++ncmds_rcvd;
407 2    ctx.ap_cmd       = c;
408 2    ctx.ap_datalen   = datalen;
409 2    ctx.ap_data      = data;
410 2    ctx.ap_resultlen = 0;
411 2    ctx.ap_resultdata = NULL;

413 2    /*
414 2     * NOTE: This switch statement is in alpha-order
415 2     *       (case-folded) to make it easier for humans
416 2     *       to find entries (at least, it was in alpha-order
417 2     *       when I wrote this comment).
418 2     */

420 2    switch (c)
421 3    {
422 3        /*
423 3         * 'p'
424 3         * Ping command.  Used for debugging.
425 3         */
427 3    case 'p':
428 3        ctx.ap_resultlen = ctx.ap_datalen;
429 3        ctx.ap_resultdata = ctx.ap_data;
430 3        break;

432 3        /*
433 3         * 'q'
434 3         * Quit command.
436 3         */
437 3    case 'q':
438 3        rbe_close_logs(logging_channel);
440 3        _exit(0);

441 3        /*
442 3         * 'r'
443 3         * rcmd filter.  Fire up a process, typically xcpiogen,
444 3         * with output from the process going to an rcmd
445 3         * connection.
447 3         */
448 3    case 'r':
449 3        z_rcmdfilter(&ctx);
451 3        break;

452 3        /*
453 3         * 'X'
454 3         * Exec separate process version of do_auxproc.
455 3         * Used for debugging.
456 3         * No value returned to parent.
458 3         * If exec fails, auxproc dies.
459 3         */
460 3    case 'X':
461 3        z_exec_separate_auxproc(&ctx, Xname);
463 3        _exit(2);
464 3        break;

465 3    default:
         rec_api_log_csm(SUB_CSM_INV_AUXPROC_CMD, NULL);
         rbe_log_stats(
             0, "auxproc -- invalid command in do_auxproc");
```

```
466 3        exit(1);
467 3        break;   /* break */
468 2    }       /* end of switch */

470 2    /*
471 2     * Simple commands set ap_resultlen and ap_resultdata
472 2     * and we push the results to the parent here.
473 2     */

475 2    if (ctx.ap_resultlen >= 0)
476 3    {
477 3        pwrite_or_die(w_fd, &c, 1, _exit);
478 3        pwrite_or_die(w_fd, (char *)&ctx.ap_resultlen,
479 3            sizeof ctx.ap_resultlen,
480 3            ctx.ap_resultdata, _exit);
481 3        pwrite_or_die(w_fd, ctx.ap_resultlen,
482 3            ctx.ap_resultlen, _exit);
483 2    }

484 2    #if 0
485 3    {
486 3        if('r' == c)
487 3            rbe_log_stats(0, "Auxproc(%d) local exit is %d.", getpid(),
                 ctx.ap_resultlen);
489 2    }
490 2    #endif
491 1    }    /* end of for loop */
492      }    /* end of do_auxproc() */
```

```
494  /*
495   * Fire up a process with stdin from parent and output to rcmd.
496   *
497   * Protocol traffic between "parent" (main process) and us:
498   *
499   *  'r' parent  --> auxproc     contains local & remote cmd info
500   *  '0' auxproc --> parent      returns "setup" result, see below
501   *  'R' auxproc --> parent      returns remote exit status (1 int)
502   *  'r' auxproc --> parent      returns local exit status  (1 int)
503   *
504   * The '0' result packet describes the results of setting things up.
505   * It contains the following information:
506   *
507   *    success/failure : int    [ 0 is success,
508   *                               non-zero is a failure code ]
509   *    errnum          : int    [ 0 if no applicable errno code ]
510   *    pid             : int    [ process ID of local xcpiogen proc
511   *                             ]
512   *    msglen          : int    [ length, in bytes,
513   *                               of following str ]
514   *    errstr          : string [ string describing failure ]
515   *
516   * The defined failure codes are:
517   *    TBD
518   *
519   * The 'R' result packet will not be sent if the '0' result
520   * indicates that an error occurred.  The '0' and 'r' result
521   * packets are always sent.
522   *
523   * If things were set up successfully, then two integer zero
524   * values, a non-zero pid, and one more zero (msglen) are
525   * sent in the '0' packet.
526   *
527   * Parent passes the following in the 'r' packet <data>:
528   *
529   *    (string)  rcmd-hostname
530   *    (string)  rcmd-locuser
531   *    (string)  rcmd-remuser
532   *    (string)  rcmd-cmd
533   *    (string)  filter-cmd
534   *    (int)     filter-cmd-argc
535   *    (string)  filter-cmd-argv0
536   *    (string)  ...
537   *    (string)  filter-cmd-argvN
538   *    (int)     setuid-val
539   *    (int)     filter-cmd-fd-info (explained below)
530   *    (int)     future-flags (flags for future hacks;
531   *                            always zero now)
532   *    (int)     db API socket info
533   *    (string)  db API socket host name
534   *    (string)  filespec from workitem
535   *
536   * If filter-cmd-fd-info is -1, then stdout on the filter cmd
537   * is set up to go to the rcmd.
538   * Otherwise, a separate file descriptor (neither stdout nor stderr)
539   * is set up to go to the rcmd, and the argv element indicated
540   * by filter-cmd-fd-info is used as a sprintf template for passing
541   * the file descriptor number to the filter process.
542   *
543   * For example, if filter-cmd-fd-info is 1, then filter-cmd-argv1
544   * should be a sprintf format string, which will be given to
545   * sprintf along with one integer to pass the file descriptor number
546   * to the filter command.
547   *
548   * Items which are (string) are '\0' terminated.
549   *
550   * Parent is responsible for obeying the built-in size limits:
```

```
555   *    no more than 100 argv strings for filter-cmd
556   *    no more than 10000 total bytes of auxproc data
557   *    no more than 100 bytes of filter-cmd-fd-info argv
558   */

    struct rcmd_pkt0_info
561   {
562 1     int failcode;
563 1     int errnum;
564 1     int pid;
565 1     int msglen;
566 1     /* variable length char string message follows */
567   };

570   /*
571    * Function to build command line prefix for adding environment
572    * variable
573    * for log truncation
    */
575   static char *
    recover_size_prefix(struct auxproc_context *cxp)
577   {
578 1     static char lbuff[128];

580 1     if (cxp->ap_config->crlfile.rotation_size != NO_ROTATION)
581 2     {
582 2         sprintf(lbuff, "EB_MAX_CLIENT_LOG_SIZE=%d; "
583 2                        "export EB_MAX_CLIENT_LOG_SIZE;",
584 1                 cxp->ap_config->crlfile.rotation_size);
585 1     }
586 2     else
587 2     {
588 1         memset(lbuff, 0, sizeof(lbuff));
590 1     }

591 1     return lbuff;
      }   /* recover_size_prefix */
```

```
593     static void
594  1  z_rcmdfilter(struct auxproc_context *cxp)
595  1  {
596  1    char      *data;
597  1    char      *rcmd_stuff[4];      /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
598  1    char      *human_name;
599  1    struct rcmd_pkt0_info pkt0;
600  1    char      *pkt0_errstr = "";
601  1    static int r_resultdata;
602  1    int       filter_cmd_argc;
603  1    int       filter_cmd_fd_info;
604  1    int       filter_cmd_fdout;
605  1    char      *filter_cmd;
606  1    char      *method;
607  1    int       dbseqno=0;
608  1    int       socket_port;
609  1    char      socket_host[100];
610  1    char      *socket_file;
611  1    char      wifilename[200];
612  1    FILE      *infofd;
613     static char readmode[] = "r";        /* items for socket info file --- */
614  1    char      cbuf[200];
615  1    char      listener_filename[EB_MAXPATHLEN];
616  1    char      *workitem;
617  1    int       rcmd_stderr = -1;
618  1    int       *rcmd_fd;
619  1    int       rcmd_fds[2];
620  1    int       i,n;
621  1    char      c;
622  1    char      *p2;
623  1    int       wait_result;
624  1    char      fdarg[100];
625  1    char      *filter_cmd_argv[100];     /* BUILT-IN LIMIT */
626  1    char      databuf[10000];            /* BUILT-IN LIMIT */
627  1    char      *filespec;                 /* BUILT-IN LIMIT */
628  1    char      sslbuff[200];
629  1    char      holdbuff[1024];            /* for command with ssl parameter */
                                               /* hold rbrclient ... command */
631  1    RBC_WORKITEM *pwi;
632  1    RBC_WORKGROUP *pwg;
633  1    boolean_ty xcpiogen_still_running;
634  1    e_errno_ty errnum;

                                               /* EDMLINK handle */
638  1    /* EDMLINK handle */
639  1    ELinkHandlePtr_ty   pELinkHandle      = NULL;

                                               /* EDMLINK objects */
641  1    ElinkTargetObjPtr_ty pELinkTargetObj  = NULL;
642  1    ElinkUserIdObjPtr_ty pELinkUserIdObj  = NULL;
643  1    ElinkCmdObjPtr_ty    pELinkCmdObj      = NULL;
644  1

                                               /* EDMLINK global options */
646  1    unsigned long ELinkOptions = 0;
647  1

                                               /* EDMLINK status */
649  1    int ELinkStatus = 0;
650  1    int ELinkStatus_errno = 0;
651  1

653  1    data = cxp->ap_data;
```

```
655  1    if (data == NULL)
656  2    {
                 /*
                  * command dispatcher did not read data for us,
                  * there was too much.  We must read it.
                  */
657  2
658  2
659  2
660  2        data = databuf;
                 pread_or_die(cxp->ap_r_fd, data, cxp->ap_datalen, _exit);
662  2    }
663  2

664  1    /*
          * extract the 4 rcmd strings.
          * [3] is the recxcpio cmd.
          */
666  1
667  1
668  1
669  1    for (i = 0; i < 4; i++)
671  1    {
672  2        rcmd_stuff[i] = data;
673  2        data += strlen(rcmd_stuff[i])+1;
674  2    }
675  1

          /*
           * extract the remote uidname to be used
           */
677  1
678  1
679  1
681  1    human_name = data;
682  1    data += strlen(human_name) + 1;

684  1    /*
          * extract the filter-cmd-fd-info
          */
685  1
686  1
688  1    memcpy(&filter_cmd_fd_info, data, sizeof (int));
689  1    data += sizeof (int);

691  1    /*
          * skip the flags
          */
692  1
693  1
695  1    data += sizeof (int);

697  1    /*
          * extract the filter command
          */
698  1
699  1
701  1    filter_cmd = data;
702  1    data += strlen(filter_cmd)+1;

704  1    memcpy(&filter_cmd_argc, data, sizeof filter_cmd_argc);
705  1    data += sizeof filter_cmd_argc;

707  1    for (i = 0; i < filter_cmd_argc; i++)
708  2    {
709  2        filter_cmd_argv[i] = data;
710  2        data += strlen(filter_cmd_argv[i])+1;
711  1    }

713  1    filter_cmd_argv[filter_cmd_argc] = NULL;

715  1    /*
          * extract db API socket info
          */
716  1
717  1
719  1    memcpy((char *)&socket_port, data, sizeof (int));
```

```
720 1    data += sizeof (int);

722 1    /*
723 1     * extract db API socket host name
724 1     */

726 1    socket_file = data;
727 1    data += strlen(socket_file) + 1;

729 1    /*
730 1     * extract filespec
731 1     */

733 1    filespec = data;
734 1    data += strlen(filespec) + 1;

736 1    /*
737 1     * extract workitem
738 1     */

740 1    workitem = data;
741 1    cxp->ap_workitem = workitem;
742 1    data += strlen(workitem) + 1;

744 1    pkt0.failcode = 0;
745 1    pkt0.errnum   = 0;
746 1    pkt0.pid      = -1;
747 1    pkt0.msglen   = -1;
748 1    pkt0.errstr   = "";

750 1    /*
751 1     * locate work item in config info
752 1     */

754 1    if (NULL == cxp->ap_config)
755 2    {
756 2      if(NULL == (cxp->ap_config = malloc(sizeof(
                        struct rbc_configs))))
757 3      {
758 3        rbe_log_stats(
               0, "Could not allocate memory in z_rcmdfilter!");
759 3        exit(1);
760 2      }
761 2      if (rbc_parse_config(
762 2          NULL /*use the default name */, &cxp->ap_config,
               RBC_PARSE_DO_NOT_PRESERVE |
               RBC_PARSE_APPLY) != 0)
764 3      {
765 3        rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
766 3        rbe_log_stats(
767 3          0, "auxproc -- Cannot parse configuration file");
768 2        exit(1);
769 1      }
         }

771 1    for (pwg = cxp->ap_config->pgrouplist; ; pwg = pwg->next)
772 2    {
773 2      if (pwg == (RBC_WORKGROUP *)NULL)
774 3      {
775 3        char csm_err_msg[256];

777 3        rbe_log_stats(
778 3          0, "\n ** No work item name \"%s\" in configuration file",
               cxp->ap_workitem);
779 3        sprintf(csm_err_msg,
```

```
780 3            "Cannot restore. Work item not in eb.cfg (
781 3             No work item for %s.)",
782 3            cxp->ap_workitem);
783 3        rec_api_log_csm(SUB_CSM_NO_WITEM_CFG, csm_err_msg);
784 3        rbe_log_stats(0, "Cannot continue without a work item");
         exit(1);

787 2      }
788 3      for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
789 3      {
790 4        if (0 == strcmp(pwi->name, cxp->ap_workitem))
791 4        {
792 3          goto gotit;
793 2        }
794 1      }
         }

     gotit:
796 1    /*
797 1     * connect by direct or rsh methods
798 1     */
799 1

     method = "rsh";

801 1    if ((method = rb_getmethod(rcmd_stuff[0], NULL, NULL)) == NULL)
802 2    {
803 2      WriteFmtStringMsg(
804 2        cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_WARNING, 0,
805 2        "Unable to get connection method to host \"%s\": "
806 2        "defaulting to \"rsh\" method\n",
807 2        rcmd_stuff[0]);
808 1      method = "rsh";
         }

810 1    /*
811 1     * now check for a work item override of the connection method
812 1     */

814 1    if (CNCTN_RSH != pwi->connection_type) /* if other than rsh */
815 2    {
816 3      switch (pwi->connection_type)
817 3      {
818 3        case CNCTN_RSH:
819 3          method = "rsh";
820 3          break;
822 3        case CNCTN_EDMLINK:
823 3          method = "edmlink";
824 3          break;
826 3        case CNCTN_SOCKET:
827 3          method = "socket";
828 3          break;
830 3        case CNCTN_DIRECT:
831 3          method = "direct";
832 3          break;
834 3        case CNCTN_NETWARE:
835 3          method = "netware";
836 3          break;
838 3        default:
839 3          method = "???";
840 3          break;
841 2      }
```

```c
843      if (debugmode)
844        {
845        rbe_log_stats(
846          0, "using connection method \"%s\" due to work item \"%s\" override",
847          method, pwi->name);
848        }

850      if (NULL != pwi && pwi->recovery_init) /* if a recovery init
                                                   command specified */
851        {
852        (void)system(pwi->recovery_init);
853        }

855      /* OSGsw34952 -- Workaround for STPclose() behavioral defect
856         If pwi->ssl_groupname is not NULL we are going to be using
857         SP for the restore.
858         Shut off the SSL (STP) exit handler.

860         Environment variable is used here so the call to
861         CDL_noatexit can be avoided, if desired.
862      */

864      if ( NULL == getenv("AUXPROC_DONOT_PERFORM_CDLNOATEXIT") )
865        {
866        (void) CDL_noatexit();
867        }

869      if ( NULL != pwi->ssl_groupname )
870        {

871  #if 0
872        (void) putenv("EDM_STP_LOGGING");
873        (void) putenv("STP_EVTLEVEL=DEBUG2");
874        (void) putenv("STP_EVTLOG_SHARE=OFF");
875  #endif

877        is_symmpath = 1;

878        }

879      if (0 == strcmp(method, "direct"))
880        {
882        /*
883         * direct connection method here
884         */

885        if (debugmode)
886          {
888          rbe_log_stats(0,"Invoking cmd with %s, %s, %s, %s\n",
889            rcmd_stuff[0], rcmd_stuff[1],
890            rcmd_stuff[2], rcmd_stuff[3]);
891          }

892        strcpy(cbuf, recover_size_prefix(cxp));

893        if (strlen(cbuf) > (size_t)0)
895          {
897          sprintf(holdbuff, "( %s %s )", cbuf, rcmd_stuff[3]);
898          rcmd_stuff[3] = holdbuff;
899          }

900        rcmd_fd = ebr_direct_rcmd(
901          &rcmd_stuff[0], cxp->ap_shelltcp_port, cxp,
903          rcmd_stuff[1], rcmd_stuff[2],
904          rcmd_stuff[3],
```

```c
           &rcmd_stderr);

905      if (debugmode)
906        {
907        rbe_log_stats(
908          0, "AUXPROC: rcmd returned fd %d, stderr fd %d\n",
909          NULL != rcmd_fd ? rcmd_fd[0] : -1), rcmd_stderr);
910        }

912      if (rcmd_fd == NULL)
913        {
914        pkt0.failcode = 2;
915        pkt0.errstr = "cannot set up remote connection";
917        goto send_0;
918        }
919      else if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
920               method, "edmlink")) )
921        {
922

924        /* variable for dealing with symm path and cross restore */

925        char sslName[CDL_HOST_LENGTH];
926        char sslGroup[CDL_STG_LENGTH];
927        char **stgNames;           /* for returned array of STG names */
928        int numGroups = 0;         /* for returned number of STG names */
929        int addIndex = 0;
930        int rc = 0;
931        boolean_ty symmPathOK = FALSE;
932
933        /*
           * rsh connection method here
           */

935        /*
936         * Set up the rcmd connection (rsh method) to the client.
937         */

939        if (! cxp->ap_have_shelltcp_port)
940          {
941          struct servent *sp;

943          /*
944           * Try to get the port number again.
945           */

947          if ((sp = getservbyname("shell", "tcp")) != NULL)
948            {
949            cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
950            cxp->ap_have_shelltcp_port = 1;
951            }
952          else
953            {
954            pkt0.failcode = 1;
955            pkt0.errstr = "shell/tcp service not found";
956            goto send_0;
957            }
958          }

960        /*
961         * check for SSL enabled work item and if present add
962         * SSL information to command string as an environment
963         * variable--but first check for cross restore and make
964         * appropriate adjustments
965         */
```

```
967      if (pwi->ssl_groupname != NULL)
968      {
969          if (!ebc_same_host(
970              rcmd_stuff[0], pwi->sysname))   /* if x-recovery */
971          {
972              /* Grab the STG entries for this client */
973              rc = CDL_getavailablegroups(
974                  rcmd_stuff[0], &numGroups, &stgNames);
975              if ((numGroups==0) || (rc < 0))
976              {
977                  char errStr[128] ;
978                  sprintf(
979     errStr, "Unable to use SymmPath for cross restore to host %s--using
                 network", rcmd_stuff[0]);
980                  rbe_user_error(0, errStr);
981                  /*
982                   * Default to network by leaving SP Flag set to
                       False.
983                   */
984              }
985              else
986              {
987                  /*
988                   * The destination client is SP enabled. First check
989                   * to see if the same STG group exists.  If we don't
990                   * find the same one we will use the first, so set
                       * that as a default
                       */
992                  symmPathOK = TRUE;
993
                     /* Initialize the index */
995                  strncpy(sslGroup, stgNames[0], CDL_STG_LENGTH-1);
996                  memset(sslGroup, 0, CDL_STG_LENGTH);
997
998                  addIndex = numGroups;
                     while (--addIndex >= 0)
999                  {
1000                     if (0 == strcmp(
                             pwi->ssl_groupname, stgNames[addIndex]))
1001                     {
                             strcpy(
                             sslGroup, stgNames[addIndex], CDL_STG_LENGTH-1);
1002                         continue;

1005                         sprintf(sslbuff,
                             "EB_SSL_RECOVER=\"%s\" ;
                             export EB_SSL_RECOVER ;
                             ",
1006                         sslGroup);
                         }
                     }
                 }
             }
1009         else   /* not a cross recovery--just normal SP */
1010         {
1011             symmPathOK = TRUE;
1012             memset(sslGroup, 0, CDL_STG_LENGTH);
1013             strcpy(
1014             sslGroup, pwi->ssl_groupname, CDL_STG_LENGTH-1);
1015             sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\" ;
1016             export EB_SSL_RECOVER ;
1017             ",
                 sslGroup);
             }
1019     }
```

```
1021     }

         if (debugmode)
         {
             sprintf(cxp->ap_error_message,
                 "Calling ELinkShell with %s, %d, %s, %s, %s",
                 rcmd_stuff[0], cxp->ap_shelltcp_port,
                 rcmd_stuff[1],
                 rcmd_stuff[2], rcmd_stuff[3]);
             rbe_log_stats(0, "%s", cxp->ap_error_message);
             cxp->ap_error_message[0] = 0;
         }

1025     strcpy(cbuf, recover_size_prefix(cxp));

1036     if (strlen(cbuf) > (size_t)0)
1038     {
1039         if ( symmPathOK == TRUE)
1040         {
1041             sprintf(holdbuff, "( %s %s )", cbuf, sslbuff, rcmd_stuff[3]);
1042             rcmd_stuff[3] = holdbuff;
1043         }
1044         else
1045         {
1046             sprintf(holdbuff, "( %s )", cbuf, rcmd_stuff[3]);
1047             rcmd_stuff[3] = holdbuff;
1048         }
1049     }
1051     else if (symmPathOK == TRUE)
1052     {
1053         sprintf(holdbuff, "( %s %s )", sslbuff, rcmd_stuff[3]);
1054         rcmd_stuff[3] = holdbuff;
1055     }
1056     /* if not Symm Path and cbuf is NULL rcmd_stuff[3] stays
                                                     unchanged */

1061     /*
1062     ** EDMLINK API
1063     */

1065     /* set the initial edmlink transport methods */
1066     ELinkOptions = ELINK_SHELL_RCMD | ELINK_SHELL_REXEC;

1068     /* enable the edmlink transport method */
1069     if( 0 == strcmp(method, "edmlink") )
1070     {
1071         ELinkOptions |= ELINK_SHELL_EDMLINK;
1072     }

1074     if( debugmode )
1075     {
1076         /* turn on edmlink debugging */
1077         ELinkOptions |= ELINK_LOGLVL_DEBUG;
1078     }

1080     /* initialize the edmlink api */
1081     pELinkHandle = ELinkInitAPI( ELinkOptions ) ;
```

```
1083  2    if( NULL == pELinkHandle )
1084  3    {
1085  3        sprintf(cxp->ap_error_message,
1086  3            "ERROR: Could not initialize the EDMLINK API.");
1088  3        WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1089  3            EDMREPROGMSG_AUXPROC_ERROR, 0,
1090  3            "%s\n", cxp->ap_error_message);
1092  3        rbe_log_stats(0, "%s", cxp->ap_error_message);
1093  3        return;
1094  2    }
1096  2    /* get new target host object */
1097  2    pELinkTargetObj = ELinkNewTargetObj( pELinkHandle,
1098  2            rcmd_stuff[0] );
1100  2    if( NULL == pELinkTargetObj )
1101  3    {
1102  3        sprintf(cxp->ap_error_message,
1103  3            "ERROR: Could not create a new EDMLINK target
                      object.");
1105  3        WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1106  3            EDMREPROGMSG_AUXPROC_ERROR, 0,
1107  3            "%s\n", cxp->ap_error_message);
1109  3        rbe_log_stats(0, "%s", cxp->ap_error_message);
1110  3        /* clean up and return */
1111  3        (void) ELinkDoneAPI( pELinkHandle );
1112  3        return;
1113  3    }
1115  2    /* get new user id object */
1116  2    pELinkUserIdObj = ELinkNewEbrUserIdObj( pELinkHandle,
1117  2            pELinkTargetObj,
1118  2            rcmd_stuff[1] );
1120  2    if( NULL == pELinkUserIdObj )
1121  2    {
1122  2        WriteFmtStringMsg(cxp-> ap_w_prog_fd,
1123  3            EDMREPROGMSG_AUXPROC_ERROR, 0,
1124  3            "Could not create a new EDMLINK user id
1125  3                object.\n" );
1126  3        /* clean up and return */
1127  3        if( NULL != pELinkTargetObj )
1129  2            (void) ELinkDestroyObj(
1130  2                pELinkTargetObj,
                        pELinkHandle );
1132  2            return;
1133  2        (void) ELinkDoneAPI( pELinkHandle );
1134  2    /* get new command object */
1135  2    pELinkCmdObj = ELinkNewCmdObj( pELinkHandle,
1137  2            pELinkTargetObj,
1138  3            rcmd_stuff[3] );
1139  3    if( NULL == pELinkCmdObj )
1140  3    {
1142  3        sprintf(cxp->ap_error_message,
1143  3            "ERROR: Could not create a new EDMLINK command
1144  3                object.");
              WriteFmtStringMsg(cxp-> ap_w_prog_fd,
                  EDMREPROGMSG_AUXPROC_ERROR, 0,
                  "%s\n", cxp->ap_error_message);
```

```
1146  3        rbe_log_stats(0, "%s", cxp->ap_error_message);
1147  3        /* clean up and return */
1148  3        if( NULL != pELinkTargetObj )
1149  3            (void) ELinkDestroyObj(
                        pELinkTargetObj,
                        pELinkHandle, pELinkUserIdObj );
1150  3        if( NULL != pELinkUserIdObj )
1151  3            (void) ELinkDestroyObj(
                        pELinkHandle,
                        pELinkHandle, pELinkUserIdObj );
1152  3        (void) ELinkDoneAPI( pELinkHandle );
1153  3        return;
1154  2    }
1156  2    rcmd_fd = rcmd_fds; /* set valid pointer */
1158  2    ELinkStatus = ELinkShell( pELinkHandle,
1159  2            pELinkTargetObj,
1160  2            pELinkUserIdObj,
1161  2            pELinkCmdObj,
1162  2            &rcmd_fd[0],
1163  2            &rcmd_stderr );
1164  2    ELinkStatus_errno = errno;
1166  2    if (debugmode)
              {
                  rbe_log_stats(
                      0, "AUXPROC: ELinkShell returned %d, errno %d, "
                          "fd %d, stderr fd %d\n", ELinkStatus,
                          ELinkStatus_errno, rcmd_fd[0], rcmd_stderr );
              }
1168  2    rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */
1169  2
1170  2    /*
1171  3    ** EDMLINK: clean up and shut down the edmlink api
1172  3    */
1173  2
1175  2    if( NULL != pELinkTargetObj )
1176  2        (void) ELinkDestroyObj( pELinkHandle, pELinkTargetObj );
1177  2    if( NULL != pELinkUserIdObj )
1179  2        (void) ELinkDestroyObj( pELinkHandle, pELinkUserIdObj );
1180  2    if( NULL != pELinkCmdObj )
1181  2        (void) ELinkDestroyObj( pELinkHandle, pELinkCmdObj );
1182  2    (void) ELinkDoneAPI( pELinkHandle );
1185  2    if( NULL != pELinkCmdObj )
1186  2        (void) ELinkDestroyObj( pELinkHandle, pELinkCmdObj );
1188  2    (void) ELinkDoneAPI( pELinkHandle );
1190  2    if (0 != ELinkStatus)
1191  3    {
1192  3        sprintf(cxp->ap_error_message,
1193  3            "cannot set up remote connection when calling "
1194  3            "ELinkShell with %s, %d, %s, %s, \'%s\'",
                      error \"%s\" (%d)",
1195  3            cxp->ap_shelltcp_port,
1196  3            rcmd_stuff[0], cxp->ap_shelltcp_port,
1197  3            rcmd_stuff[2], rcmd_stuff[3],
                  esl_strerror(
                      ELinkStatus_errno), ELinkStatus_errno);
1199  3        pkt0.failcode = 2;
1200  3        pkt0.errstr = cxp->ap_error_message;
1202  4        if (debugmode)
1203  4        {
1204  4            rbe_log_stats(
```

```
1205  3            ELinkStatus_errno, "ELinkShell failed!\n");
1207  3
1208  2            goto send_0;
1210  2        }
1211  2
1212  2    )
1213  2
1214  3    /*
1215  3     * add remainder of Symmetrix Path handshaking here
1217  3     * if the we made it through the other Symm Path tests
1218  3     */
1219  3    if (symmPathOK == TRUE)
1220  3    {
1221  3        int newfd;  /* fd for SSL sopen call */
1222  3
1223  3        /*
1224  3         * We're going to grab the short name SSL alias for the
1226  3         * long network name.  Useless unless this is a cross-restore.
1227  3         *
1228  4         * (
1229  4         * because rcmd_stuff[0] == pwi->ssl_groupname on a regular
1230  4         * restore)
1231  4         */
1232  4        memset(sslName, 0, CDL_HOST_LENGTH);
1233  4        strncpy(sslName, rcmd_stuff[0], CDL_HOST_LENGTH-1);
1234  4
1236  4        if ((strlen(rcmd_stuff[0]) >= CDL_HOST_LENGTH) &&
1237  3            (0 >= CDL_getsslhostname(sslName, rcmd_stuff[0])))
1239  3        {
1240  3            char errStr[128];
1241  3            sprintf(
1243  3            "Unable to determine the short SymmPath hostname of %s",
1244  3                    rcmd_stuff[0]);
1246  4            rbe_user_error(0, errStr);
1247  4            pkt0.failcode = 2;
1248  4            pkt0_errstr = "Cross-restore is configured to go
1249  4                           through SP, yet destination host is not"
1250  4            " properly configured in edm.conf";
1253  4
1254  3            goto send_0;
1256  3        }
1257  2
1259  2        /*
          *      * Establish the SymmPath channel
          *      */
          if ((newfd = CDL_client(rcmd_fd[0],
                                  sslName,
                                  sslGroup)) <0)
          {
              rbe_user_error(0,
                          "Unable to open an SSL listener connection
                           -- CDL_client failed");

              goto send_0;
          }

          /* We connected...Use it.. */
          rcmd_fd[0] = rcmd_fd[1] = newfd ;
          pkt0_errstr = CDL_errstr(newfd);

      }
1262  1    else if (0 == strcmp(method, "netware"))
1263  1    {
```

```
1264  2    {
1265  2        /*
1266  2         * netware connection method here
1267  2         */
1269  2
1270  2        char buf9[EB_MAXPATHLEN];
1271  2        char targetbuf[EB_MAXPATHLEN];
1272  2        char *tsa = "?";
1275  2        char *target = NULL;
1276  2
1278  2        filter_cmd_argv[filter_cmd_argc++] = "-/";  /* always full path */
1279  2        filter_cmd_argv[filter_cmd_argc] = NULL;
1280  2
1282  2        /*
1283  3         * Set up the rcmd connection (rsh method) to the client.
1284  3         */
1286  3        if (! cxp->ap_have_shelltcp_port)
1287  3        {
1288  3            struct servent *sp;
1290  3
1291  4            /*
1292  4             * Try to get the port number again.
1293  4             */
1294  4            if ((sp = getservbyname("shell", "tcp")) != NULL)
1295  4            {
1296  4                cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
1297  4                cxp->ap_have_shelltcp_port = 1;
1298  4            }
1299  4            else
1300  3            {
1301  2                pkt0.failcode = 1;
1303  2                pkt0_errstr = "shell/tcp service not found";
1305  2                goto send_0;
1306  2            }
1307  2        }
1309  2
1310  3        rcmd_fd = rcmd_fds;  /* set valid pointer */
1311  3
1312  2        /*
1314  3         * now skip over "./host/bin/startrec"
1315  3         */
1316  3        for (p2 = rcmd_stuff[3]; '\0' != *p2 && ' ' != *p2; p2++)
1317  2            ;  /* skip ./host/bin/startrec */
1319  2
1320  2        while (' ' == *p2)
1321  2            p2++;  /* advance to start of next word */
1323  2
1324  2        /*
1325  3         * if a -c target:/path then take out target:
1326  4         */
1327  4        *targetbuf = '\0';
1328  4        if (strlen(p2) >=2)
           {
               if (('-' == *p2) && ('c' == p2[1]))
               {
                   char *p3 = p2+2;
```

```
1329  4      char *p4;
1330  4      while ((' ' == *p3)
1331  5      {
1332  5          p3++;  /* advance to start of next word */
1333  4      }
1334  5
1335  4      p4 = p3;  /* save this as start of the prefix */
1336  5      for (; '\0' != *p3; p3++)
1337  5      {
1338  5          if ((((';' == *p3) &&
1339  6               ((';' == *p3) &&
1340  6                (('\"' == p3[1])) ||
1341  6               (('\"' == *p3) && ('\"' == p3[1]))))
1342  4              break;
1343  4      }
1344  5
1345  5      /*
1346  5       * now, if it's a netware cross recovery, get password, etc.  from
1347  5       * destination target -- don't use source target from pw.
1348  5       * pwi points to work item config struct backup was done under
1349  5       */
1350  6
1351  5      if (*p3 != 0)  /* if found the prefix */
1352  5      {
1353  5          char * tempChar = NULL;
1354  5          strncpy(targetbuf, p4, p3-p4);
1355  6          targetbuf[p3-p4] = '\0';
1356  6          target = targetbuf;
1357  5          if ('\"' == *p3)
1358  5          {
1359  4              p3++;
1360  3          }
1361  2          tempChar = strchr(p3+1,'\"');
1363  2          if (NULL != tempChar)
1364  2          {
1365  2              tempChar[0] = ' ';
1366  2          }
1367  2          strcpy(p4, p3+1);
1369  2      }
1370  3
1371  3      if (0 != strcmp(
1374  3          rcmd_stuff[0], pwi->sysname))  /* if x-recovery */
1375  3      {
1376  4          RBC_WORKITEM *pwi2 = pwi;
1377  4
1378  3          tsa = "?";
1380  3          if (*targetbuf != 0)
1381  4          {
1382  4              target = targetbuf;
1383  5          }
1384  5          for (pwg = cxp->ap_config->pgrouplist; NULL != pwg;
1385  6               pwg = pwg->next)
1386  6          {
1387  5              for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
1388  4              {
1389  3                  if (ebc_same_host(pwi->sysname, rcmd_stuff[0]))
1390  3                  {
                            goto gotit2;
                        }
                    }
```

```
1391  3              /*
1392  3               * if we did not find a wi for the target system,
1393  3               * put user prompting here.
1394  3               */
1395  3              if (NULL == pwi)
1397  4              {
1398  3                  pwi = pwi2;  /* restore old value if ! found */
1399  3              }
1400  2          }
1402  2
1403  2          /*
1404  2           * check for an encrypted password
1405  2           */
1406  3          if (NULL != pwi &&
1407  3              pwi->flags & WORKITEM_FLAGS_ENCRYPT_PASSWD)
1408  3          {
1409  3              sprintf(
1410  3              buf9, "r %s -tsa %s -login %s -epassword %s", p2,
1411  3                  ((target != NULL)
1412  3                  ? target
1413  3                  : ((
                        pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
1414  2                  tsa,
1415  2                  ((
                        pwi->nw_username != NULL) ? pwi->nw_username : "?"),
1416  3                  ((
                        pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
1417  3          }
1418  3          else
1419  3          {
1420  3              sprintf(
1421  3              buf9, "r %s -tsa %s -login %s -password %s", p2,
1422  3                  ((target != NULL)
1423  3                  ? target
1424  2                  : ((
                        pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
1426  2                  tsa,
1427  3                  ((
                        pwi->nw_username != NULL) ? pwi->nw_username : "?"),
1428  3                  ((
                        pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
1429  4          }
1430  4
1431  4          if (NULL != pwi && (CNCTN_RSH != pwi->connection_type))
1432  4          {
1433  4              if (CNCTN_NETWARE != pwi->connection_type)
1434  4              {
1435  4                  WriteFmtStringMsg(cxp->ap_w_prog_fd,
1436  4                      EDMREPROGMSG_AUXPROC_WARNING, 0,
1437  4                      "work item \"%s\" specifies "
                            "connection method %s but client "
                            "was installed to use the netware "
                            "method -- using netware method",
1438  4                      pwi->name,
                            ((
1439  4                      CNCTN_RSH == pwi->connection_type) ? "Rsh" :
                            ((
                            CNCTN_EDMLINK == pwi->connection_type) ? "Edmlink" :
                            ((
                            CNCTN_DIRECT == pwi->connection_type) ? "Direct" :
                            ((
                            CNCTN_SOCKET == pwi->connection_type) ? "Socket" :
                            ((
1440  4                      CNCTN_NETWARE == pwi->connection_type) ? "Netware" :
                            "???"))))));
```

```c
1441            }

1443            if (0 != pwi->connection_port) /* if port specified */
1444            {
1445                if (debugmode)
1446                {
1447                    rbe_log_stats(
1448                        0, "using socket port %d to connect to"
1449                        " host \"%s\", witem \"%s\"",
1450                        pwi->connection_port,
1451                        pwi->sysname,
1452                        pwi->name);
1453                }

1455                cxp->ap_shelltcp_port = (ushort_t)pwi->connection_port;
1456            }

1457            if (debugmode)
1458            {
1459                rbe_log_stats(
1460                    0, "Calling nwrcmd with %s, %d, %s, %s, %s\n",
1462                    rcmd_stuff[0],
1463                    rcmd_stuff[0], cxp->ap_shelltcp_port,
1464                    rcmd_stuff[1], rcmd_stuff[2], buf9);
1465            }

1467            if (debugmode)
1468            {
1469                rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */
1470            }

1471            rcmd_fd[0] = nwrcmd(&rcmd_stuff[0], cxp->ap_shelltcp_port,
1473                                rcmd_stuff[1], rcmd_stuff[2], buf9,
1474                                pwi->ssl_groupname, pwi->ssl_clientname,
1475                                &rcmd_stderr,
1476                                TRUE);

1478            rbe_log_stats(
1479                0, "AUXPROC: nwrcmd returned fd %d, stderr fd %d\n",
1480                rcmd_fd[0], rcmd_stderr);

1481            if (rcmd_fd[0] == -1)
1482            {
1483                pkt0.failcode = 2;
1484                pkt0_errstr = "cannot set up remote connection";
1485                goto send_0;
1487            }
1488        }
1489        else if (0 == strcmp(method, "socket"))
1490        {
1491            /*
1492            * socket connection method here
1493            */
1494            if (CNCTN_SOCKET != pwi->connection_type) /* if other than rsh */
1495            {
1496                WriteFmtStringMsg(cxp->ap_w_prog_fd,
1497                    EDMREPROGMSG_AUXPROC_WARNING, 0,
1492                    "work item \"%s\" specifies
1493                    connection method %s but "
1494                    "client was installed to use the
1495                    socket method -- using socket method",
1496                    pwi->name,
1497                    CNCTN_RSH == pwi->connection_type ? "Rsh" :
```

```c
1498                    CNCTN_EDMLINK == pwi->connection_type ? "Edmlink" :
1499                    CNCTN_DIRECT  == pwi->connection_type ? "Direct" :
1500                    CNCTN_SOCKET  == pwi->connection_type ? "Socket" :
1501                    CNCTN_NETWARE == pwi->connection_type ? "Netware" :
1502                    "???")))))));
1504            }

1505            /*
1506            * read file with name of WI (
1507                written by listener) to get info about client
1508            */
1509            socket_port = -1;
1510            dbseqno = 1;
1511            strcpy(wifilename, wifiledir);
1512            if (NULL == socket_file)
1513            {
1514                rbe_log_stats(
1515                    0, "No socket file name passed to auxproc\n");
1516                return;
1518            }

1519            strcat(wifilename, socket_file);

1520            /*
1521            * Get socket information from eblistend info file
1522            */
1523            errnum = eb_parse_listener_info_file (wifilename,
1524                                cbuf,
1525                                socket_host,
1526                                &socket_port,
1528                                &dbseqno,
1529                                listener_filename);

1530            if (errnum != E_SUCCESS)
1531            {
1532                /*
1533                * The routine already logged an error message, simply
1534                * return the error to caller
1535                */
1536                rbe_log_stats(0,
1537                    "Unable to read eblistend info file "
1538                    \"%s\"\n",
1540                    wifilename);
1541                return;
1542            }

1543            if (socket_port == -1)
1544            {
1546                rbe_log_stats(
1547                    0, "Invalid port field seen for client\n");
1548                return;
1549            }

1550            if (
1551                0 != pwi->connection_port) /* if port specified in work item */
1552            {
                    if (debugmode)
                    {
                        rbe_log_stats(
                            0, "using socket port %d to connect to"
                            " host \"%s\", witem \"%s\"",
                            pwi->connection_port,
                            pwi->sysname,
```

```
1553  4          }
1554
1555  3          socket_port = pwi->connection_port;
1556  2
1558  2
1559  3          rbe_log_stats(
1560  3  0, "host= %s port=%d seq#=%d\n", socket_host, socket_port, dbseqno);
1561  2
1563  2          rcmd_fd = rcmd_fds;  /* set valid pointer */
1565  2          /* We need to fix this up...
                    Modify sopen to call the right type
                    of socket (STP) */
1566  2
1568  2          rcmd_fd[0] = sopen(socket_host, 'c', socket_port);
1569  2          if (rcmd_fd[0] < 0)
1570  3          {
1571  3              rbe_log_stats(0, "Socket Open error: %d\n", rcmd_fd[0]);
1572  3              _exit(2);
1573  2          }
1575  2          /*
1576  2           * set up ONLY remote command file descriptors to talk to
1577  2           * via socket connection. DO NOT overwrite filter_cmd fd's
1578  2           */
1580  2          rcmd_fd[1]  = dup(rcmd_fd[0]);
1582  2          /* If we are connected,
                    determine if the user wants normal sockets
1583  2           * or ssl.
1584  2           */
1586  2          if (rcmd_fd[0] != -1)
1587  3          {
1588  3              if (!pwi->ssl_groupname)
1589  4              {
1590  4                  /*
1591  4                   * send OK status to client waiting on data socket
                                                    before
1592  4                   * data stream
1593  4                   */
1595  4                  sprintf(cbuf, "OK: %d socket connection made;", dbseqno);
1596  4                  (void)swrite (rcmd_fd[0], cbuf, (int)strlen(cbuf));
1597  3              }
1598  3              else /* ssl connect */
1599  3              {
1600  4                  char sslName[CDL_HOST_LENGTH];
1601  4                  char sslGroup[CDL_STG_LENGTH];
1602  4                  char **stgNames;
1603  4                  int numGroups = 0;  /* for returned array of STG names */
1604  4                  int addIndex = 0;   /* for returned number of STG names */
1605  4                  int rc = 0;
1607  4                  /* First test to see if this is a cross restore. If
1608  4                   * so we can't just use the STG group from the source
                                                    client
```

```
1609  4                   * since it may not exist on the destination client.
1610  4                   * We need to see if the destination is SP enabled and
1611  4                   * get a good STG group for that.  We'll use the same group
1612  4                   * if it is available.  If no STG group is available we will
1613  4                   * fall back to network.
1614  4                   */
1616  4                  if (!ebc_same_host(
                              socket_host, pwi->sysname)) /* if x-recovery */
1617  5                  {
1618  5                      /* Grab the STG entries for this client */
1619  5                      rc = CDL_getavailablegroups(
1620  5                          socket_host, &numGroups, &stgNames);
1621  6                      if (numGroups==0) || (rc < 0))
1622  6                      {
1623  6                          char errStr[128];
1624  6                          sprintf(
1625  6  errStr, "Unable to use SymmPath for cross restore to host %s--using
                              network", socket_host);
1626  6                          rbe_user_error(0, errStr);
1627  6                          /*
1628  6                           * Default to network and send normal
                                                    handshake.
1629  6                           *
1631  6                           * send OK status to client waiting on data
                                                    socket before
1632  6                           * data stream
                                   */
1634  5                          sprintf(cbuf, "OK: %d socket connection made;", dbseqno);
1635  5                          (void)swrite (rcmd_fd[0], cbuf, (int)strlen(
                                                    cbuf));
1636  6                      }
1637  5                      else
1638  5                      {
1639  6                          /*
1640  6                           * The destination client is SP enabled.
                                                    First check
1641  6                           * to see if the same STG group exists.
                                                    If we don't
1642  6                           * find the same one we will use the first,
                                                    so set
1644  6                           * that as a default
1645  6                           */
1647  6                          memset(sslGroup, 0, CDL_STG_LENGTH);
1648  6                          strncpy(
1649  6                              sslGroup, stgNames[0], CDL_STG_LENGTH-1);
1650  6                          /* Initialize the index */
1651  7                          addIndex = numGroups;
1652  7                          while (--addIndex >= 0)
1653  7                          {
1654  6                              if (0 == strcmp(
1656  6                                  pwi->ssl_groupname, stgNames[addIndex]))
1657  6                              {
                                         strcpy(
                                  sslGroup, stgNames[addIndex], CDL_STG_LENGTH-1);
                                         continue;
                                     }
                                    /*
                                     * We need to get a short name SSL alias if
                                     * the destination
```

```c
1658  6           * name happens to be long
1659  6           */
1661  6          memset(sslName, 0, CDL_HOST_LENGTH);
1662  6          strncpy(
1664  6              sslName, socket_host, CDL_HOST_LENGTH-1);
1665  6          if ((strlen(
1667  7                   socket_host) >= CDL_HOST_LENGTH) &&
1668  7              (0 >= CDL_getsslhostname(
1669  7                       sslName, socket_host)))
1670  7          {
1671  7              char errStr[128] ;
1672  7              sprintf(
1673  7                  errStr, "Unable to determine the short SymmPath hostname of %s",
1675  7                  socket_host);
1676  6              rbe_user_error(ffdb_errnum, errStr);
1677  6              pkt0.failcode = 2;
1678  6              pkt0.errstr = "Cross-restore is
1679  6                  configured to go through SP, yet destination host is not"
1680  6                  " properly configured in edm.conf";
1682  6              goto send_0;
1683  7          }
1684  7          if (ebsock_server_connect_ssl(sslGroup,
1685  7                                        sslName,
1686  7                                        &rcmd_fd[0],
1687  7                                        &rcmd_fd[1],
1688  6                                        socket_host,
1689  5                                        dbseqno,
1690  4                                        holdbuff) <
1691  4                                        0)
1692  5          {
1693  5              WriteFmtStringMsg(cxp->ap_w_prog_fd,
1694  5                                EDMREPROGMSG_AUXPROC_ERROR, 0,
1695  5                                /* "%s", */ holdbuff);
1697  5              return;
1698  5          }
1699  5      }
1700  5      else
1701  5      {
1702  5          /*
1703  6           * Just a normal Sp restore--nothing special to do
1704  6           */
1705  6          if (ebsock_server_connect_ssl(
1706  6                  pwi->ssl_groupname,
1707  6                  pwi->ssl_clientname,
1708  6                  &rcmd_fd[0],
1709  6                  &rcmd_fd[1],
1710  6                  socket_host,
              6                  dbseqno,
              6                  holdbuff) < 0)
              6          {
              6              WriteFmtStringMsg(cxp->ap_w_prog_fd,
              6                                EDMREPROGMSG_AUXPROC_ERROR,
              6                                0,
              6                                /* "%s",*/ holdbuff);
              4              return;
              3          }
                      }
                      return;
                  }
```

```c
1711  2          }
1713  2          /* Now that we are finally connected, dup to std err */
1714  2          rcmd_stderr = CDL_dup(rcmd_fd[0]);
1716  1      }
1717  1      else
1718  1      {
1719  2          /*
1720  2           * unknown connection method here
1721  2           */
1722  2          WriteFmtStringMsg(
1723  2              cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
1724  2              "Bad connection method \"%s\"\n", method);
1725  1          return;
1727  1      }
1728  2      WriteFmtStringMsg(
1729  2          cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
1730  2          "Could not set up progress channel from
1731  1          xcpiogen.\n");
1732  1      _exit(2);
1734  1  }
1735  2  switch (xcpiogen_pid = pkt0.pid = fork())
1736  2  {
1737  2  case -1:
1738  2      /* error */
1739  2      (void) close(xcpiogen_pipe[0]);
1740  2      (void) close(xcpiogen_pipe[1]);
1741  2      pkt0.errstr = "fork failed";
1742  2      pkt0.failcode = 3;
1743  2      pkt0.errnum = errno;
1745  2      break;
1746  2  case 0:
1748  2      /* child */
1749  2      {
1750  2          /*
1751  2           * Make our stdin be the bulk-from-parent stream.
1753  3           */
1754  3          (void)dup2(cxp->ap_r_bulk_fd, 0);
1755  3          if (debugmode)
1756  3          {
1757  3              rbe_log_stats(
1758  4                  0, "FILTER CMD: \"%s\"",
1759  4                  filter_cmd, human_name);
1760  3              for (i = 0; i < filter_cmd_argc; i++)
1761  2                  rbe_log_stats(0, "\t%s\n", filter_cmd_argv[i]);
1763  2          }
1764  2          /*
1765  2           * If no fd info passed to child,
1766  2           * then child stdout goes to the
1767  2           * rcmd process.  Otherwise, arbitrary fd is used,
1769  3           * and we pass
1770  3           * the fd number as an argument per the format given to us
             */
          if (filter_cmd_fd_info == -1)
          {
```

```
1771  3            filter_cmd_fdout = 1;
1772  3            (void)dup2(rcmd_fd[0], 1);
1773  3        }
1774  3        else
1775  3        {
1776  3            (void) sprintf(
1777  3                fdarg, filter_cmd_argv[filter_cmd_fd_info],
1778  3                filter_cmd_fdout);
1779  3            filter_cmd_argv[filter_cmd_fd_info] = fdarg;
1780  2        }

1782  3        /*
1783  3         * Send real uid string on filter_cmd_fdout to client process
1784  2         */
1786  2        i = (int)strlen(human_name);
1787  2        if ((n = looprw(
1788  3            filter_cmd_fdout, human_name, i, write_CDL_no_eintr)) != i)
1789  3        {
1790  3            rbe_log_stats(RBRECOVER_MKERR(
1791  3                errno), "Can't write %d bytes to host \"%s\", "
1792  3                "ec=%d, errno=%d, fd=%d\n", i, rcmd_stuff[0],
1793  3                                  n, errno,
                                       filter_cmd_fdout);
1795  2            _exit(2);
1796  3        }

1797  3        if ((n = looprw(
1798  3            filter_cmd_fdout, "\n", 1, write_CDL_no_eintr)) != 1)
1799  3        {
1800  3            rbe_log_stats(RBRECOVER_MKERR(errno),
1801  3                "Can't write %d bytes to host \"%s\", "
1802  2                "ec=%d, errno=%d\n",
                       1, rcmd_stuff[0],
                       n, errno,
                       filter_cmd_fdout);
1804  2            _exit(3);
1805  2        }

1806  2        (void)dup2(xcpiogen_pipe[1], 2);
1808  2        (void)close(xcpiogen_pipe[1]);

1809  2        /* Call the CDL layer so we can warn SSL that an execvp
1810  2         is coming. If this isn't an SSL socket this call is a
1811  2         no-op
1812  2         */
1814  2        (void)CDL_execvpPrep(filter_cmd_fdout);

1815  2        (void)execvp(filter_cmd, filter_cmd_argv); /* run xcpiogen */
1816  2        rbe_log_stats(RBRECOVER_MKERR(
1817  2            errno), "Can't exec \"%s\", errno=%d\n",
1818  2            filter_cmd,errno);
            _exit(1);
            break;

1822  2    default:
1823  2        /* parent */
1824  2        (void)close(xcpiogen_pipe[1]);
1825  2        xcpiogen_prog_fd = xcpiogen_pipe[0];
1826  2        break;
1827  1    }

1829  1    /* end of switch */
        /* Parent has no use for this file descriptor any more,
                                and nuking it
```

```
1830  1     * now is important so that if the child dies while the rcmd still
1831  1     * wants more input from the child, the rcmd will die too (instead of
1832  1     * hanging around hoping that maybe this parent process will supply
            * the data) */

1833  1    /* we didn't really dup these if they were SFP sockets so
            avoid the closes in that case
            */
1835  1    if (!pwi->ssl_groupname)
1836  1    {
1837  1        if (rcmd_fd[0] != -1)
1838  1            (void)close(rcmd_fd[0]);
1839  2        if (rcmd_fd[1] != -1)
1840  2            (void)close(rcmd_fd[1]);
1841  1    }
1842  3    else
1843  2    {
1844  2        /* However in the case of nwrcmd we did get a network
1845  2         socket for rcmd_stderr so we do need to issue one
1846  1         CDL_close on the one SFP socket. We can detect that
1847  2         case because rcmd_stderr will not be equal to rcmd_fd[0]
1848  1         */
1849  1        if ( 0 == CDL_isSSLsocket(rcmd_fd[0]) )
1850  1        {
1851  1            (void) CDL_close(rcmd_fd[0]);
1852  1        }
1853  1    }

1854  2    if (rcmd_stderr != rcmd_fd[0])
1855  2    {
1856  3        if (rcmd_fd[0] != -1)
1857  3        {
1858  4            /*
1859  4             * IF not a SSL socket, do the close if not
1860  4             * we close SP socket in XCPIOGEN after
1861  4             * data has been moved by calling STPexit()
1862  4             */
1863  4            rcmd_fd[0] = -1;
1864  4        }
1866  4    }

1867  5    if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
1868  5                                 method, "edmlink")) )
1869  4    {
1871  4        /*
1872  3         * Because of occasional loss of data, due to timing issues when
1873  2         * connected via edmlink,
1874  1         *    turn on keepalive. In version 4.7 this will
1876  1         * also be addressed by edmlink, directly. (OSGsw38596)
1677  2         */
1678  2        int on = 1;
1679  2        if (-1 == setsockopt( rcmd_stderr,
1880  2                              SOL_SOCKET,
1881  2                              SO_KEEPALIVE,
1882  2                              (char *)&on,
1883  2                              sizeof(on) ) )
1884  2        {
1885  2            rbe_log_stats(
1886  2                RBRECOVER_MKERR(errno),
1887  2                "Warning: setsockopt for SO_KEEPALIVE
                          failed" );
```

```
1891  2      }
1892  1
1894  2      if(0 == pkt0.failcode)
1895  2      {
1896  2          /* Lets check here to see if xcpiogen is still running.
1897  2           * The child could have exited prior to the fork.
1898  2           */
1899  2          int ChildDone_ret;
1900  2          int ChildExitStatus;
1902  2          r_resultdata = 0;                    /* "meaningless" */
1904  2          xcpiogen_still_running = TRUE;
1905  2          sleep (1);
1906  2          ChildDone_ret = ChildDone(xcpiogen_pid, &ChildExitStatus);
1907  2          if(-1 == ChildDone_ret)
1908  3          {
1909  3              rbe_log_stats(RBRECOVER_MKERR(errno),
1910  3                  "Internal error: testing to see if xcpiogen
1911  3                                            exited.");
1913  3          }
1914  3          else if (0 != ChildDone_ret)
1915  3          {
1916  3              xcpiogen_still_running = FALSE;
1917  3              r_resultdata = ChildExitStatus;
1918  3          }
1919  3
1920  1          /*
1922  1           * Send the setup failure/success packet.
1923  1           */
      send_0:
1924  2          if (pkt0.msglen < 0)
1925  2          {
1926  2              pkt0.msglen = (int)strlen(pkt0_errstr) + 1;
1928  1          }
1929  1          c = '0';
1930  1          i = (int)sizeof(pkt0) + pkt0.msglen;
1931  1          pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
1932  1          pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
1934  1          pwrite_or_die(cxp->ap_w_fd, (char *)&pkt0, sizeof pkt0, _exit);
1935  2          if (pkt0.msglen > 0)
1936  2          {
1937  2              sprintf(cxp->ap_error_message, "%s", pkt0_errstr);
1938  2              rbe_log_stats(0, "%s", cxp->ap_error_message);
1939  2              pwrite_or_die(cxp->ap_w_fd, pkt0_errstr, pkt0.msglen, _exit);
1941  1          }
1942  1          /* If the fork was successful,
                          wait for the remote exit status to come back
1943  1           * on stderr,
                          and send it to parent in an 'R' reply.  If the fork was
1944  1           * unsuccessful, there is no 'R' reply, and, furthermore,
                          the value in the
                 1        * 'r' reply is meaningless. */
1947  1          if ((xcpiogen_pid > 0) &&
1948  1              (TRUE == xcpiogen_still_running))
1949  1          {
1950  2              int remote_exitinfo;
1951  2              int Demux_ret = 0;
1952  2              Demux_ret = DemuxAuxChildren(cxp->ap_w_prog_fd,
```

```
1953  2                  rcmd_stderr,
1954  2                  rcmd_stuff[0],
1955  2                  xcpiogen_prog_fd,
1956  2                  xcpiogen_pid,
1957  2                  &remote_exitinfo);
1958  2          if(0 != Demux_ret)
1959  2          {
1960  2              /*
1961  2               *    rbe_log_stats(
1962  2               *        0, "Error monitoring Auxproc's children.");*/
                    /* error_logging */
              }
1965  2          rbe_log_stats(0, "Auxproc(%d) remote exit is %d.", getpid(),
1966  2                        remote_exitinfo);
1967  2
       #endif
1969  2          /*
1970  2           * notify the main program of remote success/failure
1971  2           */
1972  2          c = 'R';
1973  2          i = sizeof(remote_exitinfo);
1974  2          pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
1975  2          pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
1976  2          pwrite_or_die(cxp->ap_w_fd, (char *)&remote_exitinfo, i, _exit);
       #if 0
1978  2          /*
1979  2           * Is our connection method anything BUT "netware"?
1980  2           */
1982  2          if (0 != strcmp(method, "netware"))
1983  2          {
1984  2              /*** YES ***/
1985  3              /* Now wait for the exit code of the filter ( local) process
1986  3               *
1987  3               * waitpid() may return with EINTR before the child
1988  3               * proc exits, in that case, we want to continue
1989  3               * with the wait.  (fix for OSGsw15487)
                     */
1991  3              while ((waitpid(xcpiogen_pid, &wait_result, 0) == -1) &&
1992  3                     (EINTR == errno))
1993  3              {
1994  4                  /* empty while loop */
1995  3              }
1997  3              /*
1998  3               * exited while loop either because the return value of
1999  3               * waitpid is NOT -1, or the errno is NOT EINTR
2000  3               */
2001  2          }
2002  2          else
2003  3          {
2004  3              /*** IT'S NETWARE ***/
2005  3              /************************** HACK **** HACK **** HACK **** HACK **** HACK
2006  3               ************************* HACK **** HACK **** HACK **** HACK **** HACK
2007  3               ************************* HACK **** HACK **** HACK **** HACK **** HACK
2008  3               ************************* HACK **** HACK **** HACK **** HACK **** HACK
2009  3               ************************* HACK **** HACK **** HACK **** HACK **** HACK
2010  3               ************************* HACK **** HACK **** HACK **** HACK **** HACK
```

```
2011    /*
2012     * This hack is basically for Netware.
2013     *                          Netware TCP/IP close()
2014     * does not send anything back to the (
2015     *                          local)filter process
2016     * that is running as pid pkt0.pid.
2017     *                          So the "filter" process
2018     * just keeps on sending data across the network and
2019     *                          there isn't
2020     * anybody there to read it.
2021     *
2022     * Therefore,
2023     *          this "auxproc" doesn't finish until all data has
2024     * been written to the socket even though no process is
2025     *                          reading
2026     * it. However,
2027     *          the process might hang indefinitely if the network
2028     * buffer(s) become(s) full.
2029     *
2030     * This is how we will get around this (
2031     *                          kind of a hack but could
2032     * not think of a better way).
2033     *                          None of the normal TCP/IP
2034     * mechanisms seem to work with Netware.
2035     *
2036     * If the exitinfo (
2037     *                which is zero for success or non-zero for
2038     * failure) reports failure...
2039     *
2040     * After we have received the exit info from the Netware
2041     *                          client
2042     * then try to get it's "wait_result" twice with a 2
2043     *                          second pause
2044     * in between.  If we don't get it,
2045     *                we have to assume that the
2046     * process is hung and can't wind down.
2047     *
2048     * Therefore,
2049     *          let's send a SIGPIPE to the process.  I chose
2050     * SIGPIPE because this is usually trapped by our
2051     *                          software and
2052     * handled in a controlled manner.
2053     *                Just keep on trying to kill
2054     * it, sleeping for 2 seconds,
2055     *                and then see if it went away until
```

```
2057    exitinfo = waitpid(
2058        xcpiogen_pid, &wait_result, WNOHANG);
2059    while (!exitinfo)
2060    {
2061        /*
2062         * Wait a couple of seconds longer and try again
2063         */
2064        sleep(2);
2065        exitinfo = waitpid(
2067            xcpiogen_pid, &wait_result, WNOHANG);
2068        if (!exitinfo)
2069        {
2070            kill(xcpiogen_pid, SIGPIPE);       /* BLAST IT! */
2071        }
2072    }
2073 }
2074 else
2075 {
2076    /**
2077     ** We received a success return status from Novell
2079     **/
2080 }
2081    /*
2082     * waitpid() may return with EINTR before the child
2082     * proc exits, in that case, we want to continue
2083     * with the wait. (fix for OSGsw15487)
2085     */
2086    /*
2087     * exited while loop either because the return value
2088     *                          of
2089     * waitpidis NOT -1, or the errno is NOT EINTR
2090     */
2091 while ((waitpid(
2092     xcpiogen_pid, &wait_result, 0) == -1) &&
2093     (EINTR == errno))
2094 {
2095    /* empty while loop */
2097 }
2098    /*
2099     * main command loop in caller will send this result for us
2101     */
2102    /*
2103     * wait_result contains the exit status of the child
2104     * process. Use the standard macros to determine the
2105     * status of the child process and set the return value
2106     * to indicate the status accurately.
2108     */
2109 if (WIFEXITED(
2110     wait_result))              /* child proc exited normally */
2111 {
2112    /*
2114     * set the result to the exit code
2115     */
    wait_result = WEXITSTATUS(wait_result);
}
```

```
2116  2      else if (WIFSIGNALED(wait_result))
2117  3      {
2118  3          /*
2119  3           * child proc exited due to an uncaught signal, so
2120  3           * set the exit code to the signal number plus our own
2121  3           * code XG_EXIT_SIGBASE.
2122  3           */
2124  3          wait_result = WTERMSIG(wait_result) + XG_EXIT_SIGBASE;
2125  2      }
2126  2      else if (WIFSTOPPED(wait_result))
2127  3      {
2128  3          /*
2129  3           * child proc is stopped, set the exit code accordingly.
2130  3           */
2132  3          wait_result = XG_EXIT_STOPPED;
2133  2      }
2135  2      r_resultdata = wait_result;
2136  1  }
2138  1  if (rcmd_stderr != -1)
2139  2  {
2140  2      /* OSGsw34952 -- Workaround for STPclose() behavioral defect
2141  2       * Shut off the SSL (STP) exit handler
2142  2       * IF not a SSL socket, do the close if not
2143  2       * we close SP socket in XCPIOGEN after
2144  2       * data has been moved by calling STPexit()
2145  2       */
2146  2      if ( 0 == CDL_isSSLsocket(rcmd_stderr) )
2147  2      {
2148  3          (void)close(rcmd_stderr);
2149  3          rcmd_stderr = -1;
2150  2      }
2151  2      else  /* SSL socket used as stderr, so clean up
2152  2               the SP stuff */
2153  2      {
2154  3          /* SP socket was set to stderr of remote
2155  3           * commands.  Since, we call CDL_noatexit
2156  3           * when we created the SP socket we will
2157  3           * explicitly call CDL_exit() that will
2158  3           * clean up the entire SP environment */
2159  3          if ( 0 == sp_cdlexitdone )
2161  3          {
2162  4              CDL_exit();
2163  4              sp_cdlexitdone = 1;
2164  4          }
2165  3      }
2166  2  }
2167  1  /*
2169  1   * set up variables that main loop will use to send 'r' reply
2170  1   */
2171  1  cxp->ap_resultlen = sizeof r_resultdata;
2173  1  cxp->ap_resultdata = (char *)&r_resultdata;
2174  1  }  /* end of z_rcmdfilter() */
2175
```

```
2178   static enum input_states decodecookie(char *cp);

2181  1  struct cookie2state {
2182  1      char *cookie;
2183  1      enum input_states state;
2184  1  };

2186  1  static struct cookie2state c2stbl[] = {
2187  1      { REMFD_COOKIE_FD2OUT,      INSTATE_COPY_TO_STDOUT },
2188  1      { REMFD_COOKIE_FD2OUT2,     INSTATE_COPY_TO_STDOUT },
2189  1      { REMFD_COOKIE_FD2OUTEND,   INSTATE_SEARCH_PREFIX0 },
2190  1      { REMFD_COOKIE_STATUS,      INSTATE_GATHER_STATUS },
2191  1      { NULL,                     INSTATE_NG }
2192  1  };

2195  2  static enum input_states
2196  2  decodecookie(char *cp)
2197  1  {
2198  1      struct cookie2state *c2s;

2201  1      for (c2s = c2stbl; c2s->cookie != NULL; c2s++)
2202  2      {
2203  2          if (strcmp(c2s->cookie, cp) == 0)
2204  3          {
2205  3              return c2s->state;
2206  2          }
2207  1      }
2208  1      return INSTATE_SEARCH_PREFIX0;
2209  1  }  /* end of decodecookie() */
```

```
static void
sigusr1_handler(int sigvalue)
{
    if (debugmode)
    {
        rbe_log_stats(0,"\nUSR1 (ATTN) signal received!\n");
    }
    if (0 < xcpiogen_pid)
    {
        (void)kill(xcpiogen_pid, SIGTERM);
    }
    ++attn_ec;
    /* end of sigusr1_handler() */
}
```

```
static void
sigterm_handler(int sigvalue)
{
    if (debugmode)
    {
        rbe_log_stats(0,"\nTERM signal received!\n");
    }

    rbe_close_logs(logging_channel);

    /* ESS workaround: Make a call to CDL_exit
       if symmpath workitem and CDL_exit has
       not already been called.  This is to clean
       up sockets since we have made a call to
       CDL_noatexit, we have to make sure we
       clean up */

    if ( (1==is_symmpath) && (0==sp_cdlexitdone) )
    {
        CDL_exit();
        sp_cdlexitdone = 1;
    }

    signal(SIGTERM, SIG_DFL); /* restore default (terminate) action */
    (void)kill(getpid(), SIGTERM);
} /* end of sigattn_handler() */
```

```
2256    /*
2257     * Invoked when running recover in debugging mode.
2258     * This execs a separate a.out file to implement the auxproc, so that
2259     * breakpoints can be set in the recover code without affecting
2260     * the auxproc code.
2261     */
2263    static void
2264    z_exec_separate_auxproc(struct auxproc_context *cxp,
2265                            char *pathname)
2266  1 {
2267  1    char procnum_str[32];
2268  1    char r_fd_str[32];
2269  1    char w_fd_str[32];
2270  1    char r_bulk_fd_str[32];
2271  1    char dbgmodestr[32];
2272  1    char *argv0;

2274  1    /*
2275  1     * NOTE: This name is "special"; the recover main()
2276  1     * function looks for it to know when it's being invoked
2277  1     * to perform auxproc processing.
2278  1     */

2280  1    argv0 = "ebr_auxproc";

2282  1    (void)sprintf(procnum_str, "%d", cxp->ap_my_auxnum);
2283  1    (void)sprintf(r_fd_str,  "%d", cxp->ap_r_fd);
2284  1    (void)sprintf(w_fd_str,  "%d", cxp->ap_w_fd);
2285  1    (void)sprintf(r_bulk_fd_str, "%d", cxp->ap_r_bulk_fd);
2286  1    (void)sprintf(dbgmodestr, "%d", debugmode);
2287  1    (void)execlp(pathname,            /* prog to execute */
2288  1              argv0,                  /* argv 0 */
2289  1              procnum_str,            /* argv 1 */
2290  1              r_fd_str,               /* argv 2 */
2291  1              w_fd_str,               /* argv 3 */
2292  1              r_bulk_fd_str,          /* argv 4 */
2293  1              dbgmodestr,             /* argv 5 */
2294  1              (char *)0);             /* end-of-args */

2296  1    /*
2297  1     * if we get here, the exec did not go. Caller will bomb for us.
2298  1     */

2300  1 }               /* end of z_exec_separate_auxproc() */
```

```
2303    /*
2304     * Use these functions with looprw to build
2305     * a loop read (or loop write) that ignores EINTR.
2306     */
2308    static int
2309    read_CDL_no_eintr(int fd,
2310                  char *buf,
2311                  int nbytes)
2312  1 {
2313  1    int r;

2315  1    do
2316  2    {
2317  2       errno = 0;
2318  2       r = CDL_read(fd, buf, (uint_t)nbytes,0);
2319  1    } while (r == -1 && errno == EINTR);

2321  1    return r;
2322  1 }               /* end of read_CDL_no_eintr() */
```

```
2324   static int
2325   write_CDL_no_eintr(int fd,
2326                      char *buf,
2327                      int nbytes)
2328 1 {
2329 1     int r;

       do
       {
2331 1         errno = 0;
2332 2         r = CDL_write(fd, buf, (uint_t)nbytes,0);
2333 2     } while (r == -1 && errno == EINTR);
2334 2
2335 1     return r;
2337 1     /* end of write_CDL_no_eintr() */
2338   }
```

```
2342   /*
2343    * Wait, interruptibly,
2344    *                  for at least one byte to become available on fd.
2345    *
2346    * Returns 0 if at least one byte is available.
2347    * Returns -1 for any type of failure, including wait interruption.
2348    * Sets errno appropriate when -1 is returned.
       */
2350   int
2351   fd_avail_1_wait_intr(int fd)
2352   {
2353 1     esl_fdset_ty rdbits;

2356 1     E_FD_ZERO(&rdbits);
2357 1     E_FD_SET(fd, &rdbits);

2359 1     /*
2360 1      * Don't need to examine rdbits after select, since only one
2361 1      * fd is in the set -- therefore return value can be computed
2362 1      * directly from select return value.
2363 1      */
2365 1     if (esl_select(E_FD_SETSIZE, &rdbits, NULL, NULL, NULL) == -1)
2366 2     {
2367 2         return -1;
2368 1     }
2369 1     return 0;
2370       /* end of fd_avail_1_wait_intr() */
       }
```

```
2372   /*
2373    * Test, interruptibly,
                for at least one byte to become available on fd.
        *
2374    * Returns 1 if at least one byte is available.
2375    * Returns -1 for any type of failure, including test interruption.
2376    *
2377    * Return 0 if no data is available.
2378    *
2379    * Sets errno appropriate when -1 is returned.
2380    */

2382   int
2383   fd_avail_test_intr(int fd)
2384   {
2386       int retStatus;
2387       struct pollfd fd_test;

2389       fd_test.fd = fd;
2390       fd_test.events = POLLIN;
2391       fd_test.revents = 0;

2394       /*
2395        * Don't need to examine rdbits after select, since only one
2396        * fd is in the set -- therefore return value can be computed
2397        * directly from select return value.
2398        */
2400       if ((retStatus = CDL_poll_read(&fd_test, 1, 0)) == -1)
2401       {
2402           /* ERROR encountered */
2403           return -1;
2404       }
2405       return retStatus;

2407   }   /* end of fd_avail_test_intr() */
```

```
2409   /*
2410    * Test, for at least one byte to become available on fd.
2411    *
2412    * Returns 1 if at least one byte is available.
2413    * Returns -1 for any type of failure other than EINTR.
2414    *
2415    * Return 0 if no data is available.
2416    *
2417    * Sets errno appropriate when -1 is returned.
2419    */
2420   int fd_avail_test1(int fd)
2421   {
2422       int retStatus;
           struct pollfd fd_test;

2424       fd_test.fd = fd;
2425       fd_test.events = POLLIN;
2426       fd_test.revents = 0;

2429       /*
2430        * Don't need to examine rdbits after select, since only one
2431        * fd is in the set -- therefore return value can be computed
2432        * directly from select return value.
2433        */
2435       while ((-1 == (retStatus = CDL_poll_read(&fd_test, 1, 0))) &&
2436              (EINTR == errno))
2437       {
2438           ;
2439       }
2440       return retStatus;
2441   }
```

```c
#define EBR_FORK fork        /* portability definition */

static int ebr_direct_rcmd_fds[2] = { -1, -1 };

/*
 * Function to do a direct fork()/exec(
 *     ) instead of an rcmd to start up the
 * client programs.
 *     Returns ptr to in & out fds for stdio with client process.
 * Returns ptr to fds if successful,
 *     NULL if an error.  stderr from new
 * process is returned on *fd2p.
 *     Should only be called when client == server.
 */
int *
ebr_direct_rcmd(char     **ahost,
                ushort_t   inport,
                struct auxproc_context *cxp,
                char      *locuser,
                char      *remuser,
                char      *cmd,
                int       *fd2p)
{
    int     pid;
    int     x[2];
    int     y[2];
    int     z[2];
    int     fd_w;
    int     fd_r;
    int     fd_err;
    char   *client_home = NULL;
    char   *p;
    struct passwd *pw;
    struct sigaction  new_act;
    struct sigaction  old_act;
    int     istat;

    /*
     * now open pipes that will become daemons stdin, stdout, stderr
     * remember that x[0] is for read and x[1] is for write
     */
    if ((-1 == pipe(x)) || (-1 == pipe(y)) || (-1 == pipe(z)))
    {
        WriteFmtStringMsg(
            cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
            "Unable to create a pipe\n");

        return NULL;
    }

    fd_w   = x[1];    /* pipe for child's stdin */
    fd_r   = y[0];    /* pipe for child's stdout */
    fd_err = z[0];    /* pipe for child's stderr */

    /*
     * Prepare sigaction parameters
     * ignore SIGCHLD during this
     */
    sigemptyset(&new_act.sa_mask);
    new_act.sa_handler = SIG_IGN;
    new_act.sa_flags   = E_SA_RESTART;
    istat = sigaction(SIGCHLD, &new_act, &old_act);
```

```c
    if ((pid = EBR_FORK()) == 0)
    {
        /*
         * child processing goes here while parent is stopped (
         *                                        if vfork())
         * first, setup stdio to work using the pipes
         */
        if ((close(0)<0) || (close(1)<0) || (close(2)<0))
        {
            WriteFmtStringMsg(cxp->ap_w_prog_fd,
                EDMREPROGMSG_AUXPROC_ERROR, 0,
                "unable to close std(
                    in|out|err) for forked child\n");

            _exit(-1);
        }

        if ((dup(x[0])<0) || (dup(y[1])<0) || (dup(z[1])<0))
        {
            WriteFmtStringMsg(
                cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                "unable to dup pipe ends for forked
                    child\n");

            _exit(-1);
        }

        if ((close(x[0])<0) || (close(y[0])<0) || (close(z[0])<0)
            || (close(x[1])<0) || (close(y[1])<0) || (close(z[1])<0))
        {
            WriteFmtStringMsg(
                cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                "unable to close pipe ends for forked
                    child\n");

            _exit(-1);
        }

        /*
         * At this moment the process has a real UID of the user
         * executing [x]ebrecover and an effective UID of root (
         *                                    since [x]ebrecover has
         * the setuid bit set and is owned by root).
         *
         * problem 1.
         * The first problem is the access(
         *                          ) function call uses the real UID
         * of the process.  So if the user is not root or ebadmin,
         *                                          the access
         * function will fail.  This causes the recovery to fail.
         *                                          Thus
         * implementation does maintain the effective UID of the
         *                                          process.
         * is changed back to the real UID of the process.  However,
         *                                          BSD
         * to processes doing an exec of a shell.
         *                          Instead the effective UID
         * The second problem is System V Unix does not pass the
         *                                          effective UID
         * On System V,
         *     neither the real or effective UID will be root during
         * the exec of the shell script if the user executing
         *                                          [x]ebrecover is
         * a non-root user.  This process will not be able to perform
         * /bin/sh -c .
         *     /<nodename>/bin/startrec because this processes real
```

```
2551  2        * and effective UID (which are now the same) does not have
2552  2        * permissions to the directory ~ebadmin.
2553  2        *
2554  2        * The recxcpio process will be reading the username of the
2555  2        *              initiating
2556  2        * process from the standard input and setting the real UID to
2557  2        *              the
2558  2        * UID of that user so it will do no harm to set the real UID
2559  2        *              to root
2560  2        * at this point.
2561  2        *              It will be changed back to the actual user as one
2562  2        * of the first things in recxcpio.
2563  2        */
2564  2       (void)setuid(geteuid());

2565  2       /*
2566  2        * now chdir to the client code home location
2567  2        */
2568  2       if (NULL == (pw = getpwnam(
2569  3                       remuser)))    /* lookup client home dir */
2570  3       {
2571  3           p = "Can't get home directory";
2572  2           goto default_home;
2573  2       }
2574  3       else
2575  3       {
2576  4           if ((
2577  4               client_home = pw->pw_dir) != NULL) /* if there is a pointer */
2578  5           {
2579  5               if (0 != chdir(client_home))     /* cd to directory */
2580  4               {
2581  4                   p = "Can't chdir to home";
2582  3                   goto default_home;
2583  3               }
2584  4           }
2585  4           else
2586  4           {
2587  4               p = "Can't figure out home dir";

          default_home:
2588  4               WriteFmtStringMsg(cxp->ap_w_prog_fd,
2589  4                   EDMREPROGMSG_AUXPROC_WARNING, 0,
2590  4                   "%s:
2591  4   \"%s\" for user \"%s\", defaulting to /usr/epoch/EB\n",
2592  3                   p, (
2593  2                   client_home == NULL) ? "??" : client_home, remuser);
2594           client_home = "/usr/epoch/EB/CLIENT_HOME";
2595  2           }
2596  2       }

2597  2       /*
2598           * client_home now points to the home dir of the client
2599  2        *              software
2600  2        */

2601  2       /*
2602  2        * Make sure that the home directory that we finally ended
2603  2        *              up with
2604  2        * is really there.
```

```
          * Yea.. I know.
          *              I may have already done this if I found a passwd
          * entry for the client backup username and it had a home
          *              directory.
          *
          * However,
          *              I have to do it again just in case I came from another
          * path.
          *              I did not really feel like trying to fix up the spagetti
          * code. I'm feeling a little lazy today...
          */
2605  2       if (0 != chdir(client_home))
2606  2       {
2607  2           WriteFmtStringMsg(
2608  2               cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2610  2               "Server backup directory \"%s\" not found
2611  3                   or not searchable\n",
2612  3               client_home);

2613  3           _exit(-1);
2614  3       }

2615  3       (void)execl("/bin/sh", "sh", "-c", cmd, 0);
2616  2       WriteFmtStringMsg(
2618  2           cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2619  2           "Unable to execl /bin/sh sh -c %s\n", cmd);

2620  2       _exit(127);
2621  2   }

2622  1   /*
2624  1    * parent code resumes here
2625  1    */

2626  1   /*
2628  1    * Close the parent's copies of the child-ends of the pipes
2629  1    */
2630  1   close(x[0]);                    /* pipe for child's stdin  */
2631  1   close(y[1]);                    /* pipe for child's stdout */
2632  1   close(z[1]);                    /* pipe for child's stderr */

2636  1   /*
2637  1    * check for fork() failure
2638  1    */
2640  1   if (-1 == pid)
2641  2   {
2642  2       WriteFmtStringMsg(
2643  2           cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
2645  2           "fork() failed\n");

2646  2       close(x[1]);                /* pipe for child's stdin  */
2647  2       close(y[0]);                /* pipe for child's stdout */
2648  2       close(z[0]);                /* pipe for child's stderr */
2649  3       if (istat == 0)
2650  2       {
2651  2           sigaction(SIGCHLD, &old_act, NULL);
2652  2       }
2653  1       return NULL;
2655  1   }

2656  1   /*
2657  1    * now fill in fd's to be returned
2659  1    */
2660  1   ebr_direct_rcmd_fds[0] = fd_w;
2661  1   ebr_direct_rcmd_fds[1] = fd_r;
2662  2   if (NULL != fd2p)
2663  2   {
2664  1       *fd2p = fd_err;
```

```
2666 1      if (istat == 0)
2667 2      {
2669 2          sigaction(SIGCHLD, &old_act, NULL);
2669 1      }
2671 1      return ebr_direct_rcmd_fds;
2672   }    /* end of ebr_direct_rcmd() */
```

```
2674     /*
2675      * routine to search the clients_installed file for a clients
                connection
2676      * method.  Returns a pointer to the connection method string if
                successful,
2677      * NULL if the client entry could not be found.
2678      */
2680     static char *
2681     rb_getmethod(register char *host,
2682                    uint_t *concurrency,
2683                    uint_t *client_type)
2684     {
2685       FILE              *file_ptr;
2686       FILE              *lock_ptr;
2687       FFDB_HOLDER       *inclient_holder;
2688       ffdb_inclient_ty  *installed_client;
2689       eperrno           errno;
2690       static char       rb_ci_lastmethod[32] = "";

2692     #ifdef PARAM_CHECK
2693       if (NULL == host)
2694       {
2695         rbe_internal_error(RBRECOVER_MKERR(
                                   EINVAL), null_arg, "host name");
2696         return NULL;
2697       }
2698     #endif /* PARAM_CHECK */

2700       /*
2701        * get pathname to file
2702        */
2704       if (ffdb_inclient_path == NULL) /* if file name needs to be
                                              prepared */
2705       {
2706         if (ffdb_inclient_init(FFDB_INIT_DEFAULT_PATH) != 0)
2707         {
2708           return NULL;
2709         }
2710       }

2712       /*
2713        * request was not in cache -- go load it
2714        */
2716       if ((lock_ptr = ffdb_lock_file(ffdb_inclient_path,
2717                                      "restore scanning
                                         clients_installed file",
                                         1, NULL, 0)) == NULL)
2718       {
2719         (void)rbe_user_error(
2720            ffdb_errnum,
                    "Unable to obtain lock on \"%s\" for scanning",
                    ffdb_inclient_path);
2722         return NULL;
2723       }
2725       if ((file_ptr = ffdb_open_file(ffdb_inclient_path)) == NULL)
2726       {
2727         (void)rbe_user_error(
2728            ffdb_errnum, "Unable to open file \"%s\" for scanning",
                    ffdb_inclient_path);
2729         ffdb_unlock_file(lock_ptr);
2730         return NULL;
2731       }
```

```
2733 1      while ((inclient_holder = ffdb_read_record(
2734 2                      file_ptr, &errnum)) != NULL)
2735 2      {
2736 2          installed_client = (ffdb_inclient_ty *)inclient_holder->drec;
2737 3          if (errnum == 0)
2738 3          {
2739 4              if (ebc_same_host(
2740 4                  host, installed_client->hostname)) /* a match? */
2741 5              {
2742 5                  if (concurrency != NULL)
2743 6                  {
2744 6                      if (0 == installed_client->concurrency)
2745 6                          *concurrency = 32767;
2746 5                      else
2747 6                          *concurrency = installed_client->concurrency;
2748 5                  }
2749 4                  if (client_type != NULL)
2750 4                  {
2751 4                      *client_type = installed_client->platform;
2752 5                  }
2753 5                  strcpy(rb_ci_lastmethod, installed_client->method);
2754 5                  (void)ffdb_close_file(file_ptr);
2755 4                  ffdb_unlock_file(lock_ptr);
2756 4                  ffdb_destroy_holder(inclient_holder);
2757 4                  return rb_ci_lastmethod;
2758 4                                          /* return allocated string */
2759 4              }
2760 3          }
2761 2          ffdb_destroy_holder(inclient_holder);
2762 2      }
2763 2
2765 1      /*
2766 1       * if we get here -- did not find host in clients_installed file
2767 1       */
2769 1      (void)ffdb_close_file(file_ptr);
2770 1      ffdb_unlock_file(lock_ptr);
2771 1      return NULL;
2772         /* end of rb_getmethod() */
```

```
2775     /*
2776      * Reads remote file descriptor for reads of stderr
2777      * of the remote cmd. Read is done until nothing is left
2778      * then returns with or without exit code.
2779      * The error/warning messages come first and a dealt with,
2780      * if the exit code of the remote cmd is returned.
2781      *
2782      * (Inp) int fd -- The file descriptor to wait on and read.
2783      * (Out) int *exitp -- The Exit code.
2784      * (Inp) char *remhostname -- The remote client name.
2785      * (Inp) boolean_ty first_call -- Is this the first call ??
2786      * (I/O) enum input_states *state_ptr
2787      * (I/O) enum input_states *next_state_ptr
2788      * (I/O) boolean_ty *skipping_leading_whitespace.
2789      * (I/O) int *parsePos the parse position of the messages.
2790      * (I/O) int *msgPos the position of the logging message.
2791      *
2792      * None of the I/O vars need to be initialized for the first call.
2793      *
2794      * Returns: boolean_ty
2795      * False: The remote command has not exitted.
2796      * True: The remote command has exitted.
2797      */
2798     static boolean_ty
2799     parse_remote_stderr_info2(int prog_fd,
2800                               int remote_fd,
2801                               int *exitp,
2802                               char *remhostname,
2803                               boolean_ty first_call,
2804                               enum input_states *state_ptr,
2805                               enum input_states *next_state_ptr,
2806                               boolean_ty *skipping_leading_whitespace,
2807                               int *parsePos,
2808                               int *msgPos)
         {
2810 1      enum input_states previous_state;
2811 1      static char cookiebuf[REMFD_MAX_COOKIE_LEN+1];
2812 1      int protfailed = 0;
2813 1      int done = 0;
2814 1      char c;
2815 1      int n;
2816 1      #define MSGBUFFLEN 260
2817 1      static char msglogBuff[MSGBUFFLEN + 1];
2818 1      boolean_ty ret_status = FALSE;
2819 1      static int temp_exit_status;
2820 1      int write_prog = 1;
2821 1
2823 1      *exitp = 0; /* Initialize to success. */
2825 1      if (TRUE == first_call)
2826 2      {
2827 2          /* First time processing. */
2829 2          if (debugmode)
2830 3          {
2831 3              rbe_log_stats(
2832 3                  0,
2833 3                  "AUXPROC: Processing stderr info from fd %d\n",
                       remote_fd);
2835 2          }
2835 2          *state_ptr = INSTATE_NG;
2836 2          *next_state_ptr = INSTATE_SEARCH_PREFIX0;
2837 2          *parsePos = 0;
```

```
2838  2        *exitp = 0;
2839  2        *msgPos = 0;
2840  2        temp_exit_status = 0;
2841  2        *skipping_leading_whitespace = FALSE;
2842  2        n = 0;
2843  1      }
2844  1      else
2845  2      {
2846  2        n = *parsePos;
2847  1      }

2849  1      while (!protfailed && !done)
2850  2      {
2851  2        int start_ec;
2852  2        int r;

2855  2        /*
2856  2         * Wait for the next character from the remote
2857  2         * stderr stream.  Ignore interrupts, unless
2858  2         * they were due to the attention signal.
2859  2         */

2861  2        start_ec = attn_ec;
2862  2        do
2863  3        {
2864  3          r = fd_avail_test_intr(remote_fd);
2865  3          if (0 == r)
2866  4          {
2867  4            return ret_status;
2868  3          }
2869  2        } while (r == -1 && errno == EINTR && attn_ec == start_ec);

2871  2        /*
2872  2         * now that there is a character, read it
2873  2         */
2874  2        c = 0;

2876  2        if (r != -1)
2877  3        {
2878  3          do
2879  4          {
2880  4            r = CDL_read(remote_fd, &c, 1, 0);
2881  3          } while((-1 == r) && (errno == EINTR) && (
                          attn_ec == start_ec));
2882  2        }

2884  2        if (r != 1)
2885  3        {
2886  3          *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2887  3          protfailed = 1;
2888  3          ret_status = TRUE;
2889  3          break;
2890  2        }

2892  2        previous_state = *state_ptr;
2893  2        *state_ptr = *next_state_ptr;

2895  2        switch (*state_ptr)
2896  3        {
2897  3          case INSTATE_SEARCH_PREFIX0:
2898  3            if (debugmode)
2899  4            {
2900  4              rbe_log_stats(0, "AUXPROC: (
                        %c) INSTATE_SEARCH_PREFIX0\n", c);
2901  3            }
```

```
2903  3            if (c == REMFD_MAGIC_PREFIX[0])
2904  4            {
2905  4              *next_state_ptr = INSTATE_SEARCH_PREFIXN;
2906  4              n = 1;
2907  4              *parsePos = 1;
2908  3            }
2909  3            break;

2911  3          case INSTATE_SEARCH_PREFIXN:
2912  3            if (debugmode)
2913  4            {
2914  4              rbe_log_stats(0, "AUXPROC: (
                        %c) INSTATE_SEARCH_PREFIXN\n", c);
2915  3            }

2917  3            if (c != REMFD_MAGIC_PREFIX[n])
2918  4            {
2919  4              *next_state_ptr = INSTATE_SEARCH_PREFIX0;
2920  3            }
2921  3            else
2922  4            {
2923  4              n++;
2924  4              (*parsePos)++;
2925  5              if (n == REMFD_MAGIC_LENGTH)
2926  5              {
2927  5                *next_state_ptr = INSTATE_GATHER_COOKIE;
2928  5                n = 0;
2929  5                *parsePos = 0;
2930  4              }
2931  3            }
2932  3            break;

2934  3          case INSTATE_GATHER_COOKIE:
2935  3            if (debugmode)
2936  4            {
2937  4              rbe_log_stats(0, "AUXPROC: (
                        %c) INSTATE_GATHER_COOKIE\n", c);
2938  3            }

2940  3            if (c == REMFD_MAGIC_SUFFIX[0])
2941  4            {
2942  4              /*
2943  4               * Got the cookie
2944  4               */

2946  4              cookiebuf[n] = '\0';
2947  4              *next_state_ptr = INSTATE_SEARCH_SUFFIXN;
2948  4              n = 1;
2949  4              *parsePos = 1;
2950  4            }
2951  4            else if (strchr(REMFD_COOKIE_CHARS, c) == NULL)
2952  4            {
2953  4              /*
2954  4               * We found a valid prefix (else we would
2955  4               * not be here) but the cookie contains an
2956  4               * illegal character.  Should not happen;
2957  4               * there has been some protocol failure.
2958  4               */

2960  4              *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2961  4              ret_status = TRUE;
2962  4              protfailed = 1;
2963  3            }
2964  3            else if (n == REMFD_MAX_COOKIE_LEN)
2965  4            {
```

```
2966  4              /*
2967  4               * Should have seen the SUFFIX[0] by now.
2968  4               */
2970  4              *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2971  4              ret_status = TRUE;
2972  4              protfailed = 1;
2973  3          }
2974  3          else
2975  4          {
2976  4              /*
2977  4               * another cookie character.
2978  3               */
2980  4              cookiebuf[n] = c;
2981  4              n++;
2982  4              (*parsePos)++;
2983  3          }
2984  3          break;

2986  3      case INSTATE_SEARCH_SUFFIXN:
2987  3          if (debugmode)
2988  4          {
2989  4              rbe_log_stats(0, "AUXPROC: (
                        %c) INSTATE_SEARCH_SUFFIXN\n", c);
2990  3          }

2992  3          if (c != REMFD_MAGIC_SUFFIX[n])
2993  4          {
2994  4              *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2995  4              ret_status = TRUE;
2996  4              protfailed = 1;
2997  3          }
2998  3          else
2999  4          {
3000  4              n++;
3001  3              (*parsePos)++;
3002  4              if (n == REMFD_MAGIC_LENGTH)
3003  5              {
3004  5                  *next_state_ptr = INSTATE_NEWLINE;
3005  4              }
3006  3          }
3007  3          break;

3009  3      case INSTATE_GATHER_STATUS:
3010  3          if (debugmode)
3011  4          {
3012  4              rbe_log_stats(0, "AUXPROC: (
                        %c) INSTATE_GATHER_STATUS\n", c);
3013  3          }

3015  3          if (isdigit(c))
3016  4          {
3017  4              char xx[2];

3019  4              xx[0] = c;
3020  4              xx[1] = '\0';

3022  4              temp_exit_status *= 10;
3023  4              temp_exit_status += atoi(xx);
3024  4              n++;
3025  4              (*parsePos)++;
3026  4              *skipping_leading_whitespace = FALSE;
3028  3          }
3029  3          else if (c == '\n')
```

```
3030  4          {
3031  4              *exitp = temp_exit_status;
3032  4              ret_status = TRUE;
3033  4              done = 1;
3034  3          }
3035  3          else if (c == ' ' &&
3036  3                  ((n == 0) || *skipping_leading_whitespace ))
3037  4          {
3038  4              *skipping_leading_whitespace = TRUE;
3040  4          }
3041  4          else
3042  4          {
3043  3              /*
3044  3               * no-op, skip leading whitespace
3045  4               */
3046  4              *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
3047  4              ret_status = TRUE;
3048  3              protfailed = 1;
3049  3          }
3050  3          break;

3052  3      case INSTATE_COPY_TO_STDOUT:

3054  3          /*
3055  3           * Eventually, the protocol should include
3056  3           * an explicit length for this state, so
3057  3           * we can do large read/writes and so we will not
3058  3           * be vulnerable to confusion based on gunk being
3059  3           * copied to stdout.
3060  3           *
3061  3           * For now, just shove the characters at stdout
3062  3           * and stop as soon as we fix PREFIX[0]
3063  3           */

3065  3          if (c != REMFD_MAGIC_PREFIX[0])
3066  3          {
3067  3              /*
3068  4               * build a buffer to write to the restore log
3069  4               */
3070  4              if ((('\n' == c) || ((*msgPos) >= MSGBUFFLEN))
3071  5              {
3072  5                  msgLogBuff[*msgPos] = '\0';
3073  5                  (void) rbe_log_stats(0,
3074  5                          "%s: %s",
3075  5                          remhostname,
3076  5                          msgLogBuff);
3077  5                  if (write_prog)
3078  6                  {
3079  6                      WriteStringMsg(prog_fd,
3080  6                          EDMREPROGMSG_RESTORE_RCMD_UNKNOWN,
3081  6                          0,
3082  6                          msgLogBuff);
3083  5                  }
3085  5                  memset(msgLogBuff, 0, MSGBUFFLEN + 1);
3086  5                  *msgPos = 0;
3087  4              }
3088  4              else
3089  4              {
3091  5                  msgLogBuff[*msgPos] = c;
3092  5                  (*msgPos)++;
3093  4              }
```

```
3095 3            else
3096 3            {
3097 4                n = 1;
3098 4                *parsePos = 1;
3099 4                *next_state_ptr = INSTATE_SEARCH_PREFIXN;
3100 4
3101 4            break;
3102 3            }
3102 3        break;

3104 3        case INSTATE_NEWLINE:
3105 3            if (c == '\n')
3106 4            {
3107 5                if (previous_state == INSTATE_SEARCH_SUFFIXN)
3108 5                    *next_state_ptr = decodecookie(cookiebuf);

3110 5                memset(cookiebuf, 0, REMFD_MAX_COOKIE_LEN+1);

3112 5                n = 0;
3113 5                *parsePos = 0;
3114 4            else
3115 4            {
3116 5                *next_state_ptr = INSTATE_SEARCH_PREFIX0;

3117 5            }
3118 4            break;
3119 3            }
3120 3        break;
3121 2        }    /* end of switch */
3122 1    }    /* end of while loop */

3124 1    return ret_status;

3126   }    /* end of parse_remote_stderr_info2() */
```

```
3131    /*
3132     * ForwardXcpiogenProgress()
3133     *
3134     * REP: Restore Engine Process, AP: auxproc, XC: xcpiogen.
3135     * This routine read progress messages from XC and forwards
3136     * then to REP. This routine should not block on a read().
3137     *
3138     * Args:
3139     *    int xcpiogen_prog_fd -- the progress channel from XC to AP.
3140     *    int restore_engine_prog_fd -- the progress channel from AP to REP.
3141     *
3142     * Returns:
3143     *    int -- the number of forwarded messages. -1 for an error.
3144     *
3145     * Side Effects:
3146     *    Read and writes file descriptors.
3147     */
3148    static int
3149    ForwardXcpiogenProgress(int xcpiogen_prog_fd,
3150                            int restore_engine_prog_fd,
3151                            boolean_ty *zero_byte_read)
3152 1  {
3153 1      int bytes_skipped = 0;
3154 1      int read_ret, write_ret;
3155 1      int fd_test;
3156 1      int msg_count = 0;
3157 1      prog_chan_msg xcpiogen_msg;
3158 1      int debug_prog = 0;
3159 1      int write_prog = 1;
3160 1
3160 1      *zero_byte_read = FALSE;
3162 1
3162 1      memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
3164 1
3164 1      while(1 == (fd_test = fd_avail_test1(xcpiogen_prog_fd)))
3165 2      {
3166 2          read_ret = ReadMsg(
3166                  xcpiogen_prog_fd, &xcpiogen_msg, &bytes_skipped);
3168 2
3168 2          if(0 != bytes_skipped)
3169 3          {
3170 3              (void) rbe_log_stats(0,
3171                      "The progress channel from xcpiogen had %d
3172                       extraneous bytes.");
3172 2          }
3173 2
3173 2          if(0 == read_ret)
3174 3          {
3174 3              *zero_byte_read = TRUE;
3175 3              break;
3176 3          }
3176 2          else if(-1 == read_ret)
3177 2          {
3177 3              break;
3179 3          }
3180 3
3180 2          if(debug_prog)
3181 2          {
3181 3              (void) rbe_log_stats(0,
3182                       "%s: %s",
3183                       "debug progress",
3184                       DEBUG_PROG_MESSAGE_PEAK(&xcpiogen_msg));
3185 3          }
3186 3
3186 2          if(write_prog)
3187 2          {
```

```
3191  3          write_ret = WriteMsg(restore_engine_prog_fd, &xcpiogen_msg);
3192  2      }
3193  2      msg_count++;
3194  2      free(xcpiogen_msg.pcm_body);

3196  1      memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
3197  1      }
3198  1      if(-1 == fd_test)
3199  2      {
3200  2          (void) rbe_log_stats(RBRECOVER_MKERR(errno),
3201  2                  "Encountered error testing xcpiogen file
3202  2                          descriptor.");
3203  1      }
3204  1      return -1;
3205  1  }
            return msg_count;
```

---

```
3207    /*
3208     * static int DemuxAuxChildren()
3209     *
3210     * This function handles IPC communications between the following
3211     * processes: REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
3212     * and RC: remote command. The RP sends error and warning messages
3213     * to AP. When the RP is finished, RC's exit status is sent last.
3214     * The remote exit status indicates that the restore is finished.
3215     * RP error and warning messages are logged and forward on as
3216     * progress messages to RP. At the same time XC will send progress
3217     * informaton to AP, and AP will forward them to REP. XC does its
3218     * own logging.
3219     *
3220     * Args:
3221     * int progress_fd -- (In) this file descriptor is from AP to REP.
3222     * int remote_fd -- (In) this file descriptor is from RC to AP.
3223     * char *remote_progname -- (In) this the RP executable name.
3224     * int xcpiogen_fd -- (In) this file descriptor is from XC to AP.
3225     * int xcpiogen_pid -- (In) the pid for XC.
3226     * int *remote_exit -- (Out) the remote exit interperated.
3227     *
3228     * REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
3229     *                                          RC: remote command.
3230     *
3231     * Returns: ??
3232     *
          */
3234    static int
3235    DemuxAuxChildren(int progress_fd,
3236                     int remote_fd,
3237                     char *remote_progname,
3238                     int xcpiogen_fd,
3239                     int xcpiogen_pid,
3240                     int *remote_exit)
3241  1 {
3243  1      int remote_exitinfo;
3244  1      int forward_ret;
3245  1      boolean_ty remote_exitted = FALSE;
3246  1      struct pollfd monitor_fds[2];
3247  1      int poll_ret;
3248  1      boolean_ty remote_first_call = TRUE;
3249  1      int logging_channel = 0;
3250  1      int start_ec = attn_ec; /* attn_ec global */
3251  1      boolean_ty xcpiogen_zero_byte_reads;
3252  1      int number_fds;
3253  1      /* The below args are needed by and maintained by
3254  1       * parse_remote_stderr_info2(). No need to initialize
3255  1       * or test them, they simply maintain state information
3256  1       * for multiple invocations of parse_remote_stderr_info2().
3257  1       */
3259  1      enum input_states remote_state_ptr;
3260  1      enum input_states remote_next_state_ptr;
3261  1      boolean_ty skip_whitespace;
3262  1      int parsePos;
3263  1      int MsgPos;

3265  1      number_fds = 2;
3266  1      monitor_fds[0].fd = remote_fd;
3267  1      monitor_fds[0].events = POLLIN;
3268  1      monitor_fds[0].revents = 0;
```

```c
3270  1        monitor_fds[1].fd = xcpiogen_fd;
3271  1        monitor_fds[1].events = POLLIN;
3272  1        monitor_fds[1].revents = 0;

3274  1        while((FALSE == remote_exited) &&
3275  1            ((0 == ( poll_ret = CDI_poll_read(
3276  1                monitor_fds, number_fds, 5000))) ||
3277  1            ((EINTR == errno) && (-1 == poll_ret) &&
3278  2                start_ec != attn_ec))))
3279  2        {
3280  3            if (0 == poll_ret)
3281  3            {
3282  3            /* timed out */
3283  3            /* check for the attention signal and other periodic
                      checks. */

3284  4            if (start_ec != attn_ec)
3285  4            {
3286  4            /* Timeout on read, AND we were canceled (SIGUSR1) --
3287  4                If this is an SSL socket read timeout, the client may
3288  4                have gone away without us knowing it,
3289  4                so break out of the loop
3290  4                without the remote exit status.
3291  4                Do this for all remote
3292  4                remote process connections,

3294  4                since the xcpiogen to recxcpio
                        connection might be SSL,

3296  4                but not the remote_fd. In this case
3297  4                the remote process will not know that xcpiogen is
3298  4                gone.
3299  4            */

3301  3            *remote_exit = SPEXIT_REMOTE_NO_STATUS;
                                /* leaving w/o status */

3303  3            rbe_user_error (0,
3304  3                    "No remote exit status received in 5
                           seconds -- stopped "
3305  3                    "waiting for aborted restore's remote
                           process to finish");

3307  2            break;
3308  2            }
3309  3            }
3310  3            else if (-1 == poll_ret)
3311  3            {
3312  3            /* eintr */
3313  3            /* log this here */
3314  3            /*rbe_log_stats(errno, "auxproc -- ");*/
3315  4            }
3316  4            else
3317  4            {
3318  4            /* Xcpiogen process is expected to send progress
3319  4             * messages,
                       including warnings and errors along with restore
                     * progress over its stderr channel.
                       We will check xcpiogen
                     * prior to the remote channel.
                       This presumes that the remote
                     * channel determines whether we are done with the
                       restore.

                  if((POLLRDNORM & monitor_fds[1].revents) ||
                     (POLLRDBAND & monitor_fds[1].revents) ||
                     (POLLIN & monitor_fds[1].revents) ||
                     (POLLHUP & monitor_fds[1].revents))
                  {
```

```c
3320  4            */
3321  4            monitor_fds[1].revents = 0;

3323  4            xcpiogen_zero_byte_reads = FALSE;

3325  4            forward_ret = ForwardXcpiogenProgress(
3326  4                xcpiogen_prog_fd,
3327  4                progress_fd,

3329  5                &xcpiogen_zero_byte_reads);
3330  5            if(TRUE == xcpiogen_zero_byte_reads)
3331  5            {
3332  5            /* Lets no longer wait on this fd,
3333  5             * if we get a zero byte read. The
3334  5             * second fd is for xcpiogen.
3335  5             */
3336  3            number_fds = 1;
3337  3            }
3338  3            }

3339  3            if((POLLRDNORM & monitor_fds[0].revents) ||
3340  3               (POLLRDBAND & monitor_fds[0].revents) ||
3341  3               (POLLIN & monitor_fds[0].revents) ||
3342  4               (POLLHUP & monitor_fds[0].revents))

3344  4            {
3345  4            /* POLLRDHUP Why do we need these conditions. */
3346  4            /* Remote process is expected to send information back
                      on stderr
3347  4             * stream. Read that information, parse it,
                      and if available
3348  4             * send remote exit status back to parent.
                      If the exit status
3349  4             * is read this loop is terminated.
                    */

3351  4            monitor_fds[0].revents = 0;
3352  4            remote_exitted = parse_remote_stderr_info2(progress_fd,
3353  4                    remote_fd,
3354  4                    &remote_next_state_ptr,
3355  4                    &remote_exitinfo,
3356  4                    &remote_progname,
3357  4                    remote_first_call,
3358  1                    &remote_state_ptr,
3359  4                    &skip_whitespace,
3360  4                    &parsePos,
3361  4                    &MsgPos);
3362  3            remote_first_call = FALSE;
3363  2            }
3364  1            }

3366  1            /* Should we test xcpiogen for a last time before we return! */

3368  1            if(1 == fd_avail_test(xcpiogen_prog_fd))
3369  2            {
3370  2            xcpiogen_zero_byte_reads = FALSE;

3372  2            forward_ret = ForwardXcpiogenProgress(xcpiogen_prog_fd,
```

```
3373 2                          progress_fd,
3374 2                          &xcpiogen_zero_byte_reads);
3375 1       }

3377 1       if (-1 == poll_ret)
3378 2       {    /* Not eintr */
3379 2          rbe_log_stats(RBRECOVER_MKERR(errno),
3380 2             "Auxproc failed to poll Children pids!");
3381 2          return -1;
3382 1       }
3383 1       if(TRUE == remote_exitted)
3384 2       {
3385 2          *remote_exit = remote_exitinfo;
3386 2          return 0;
3387 1       }
3389 1   } /* DemuxAuxChildren() */
```

# D12

```
2    /***********************************************
3    **                                          *****
4    ** File Name:   RSLinitfin.c                 *****
5    **                                          *****
6    ** Copyright (c) 1998,1999 by EMC Corporation. *****
7    **                                          *****
8    ** Purpose:                                  *****
9    **                                          *****
10   **     This module contains the Restore Service Library
11   **     functions to
12   **     initialize and terminate the restore operation.
13   **
14   ** Table of Contents:
15   ** ------------------
16   **     RSTSL_Initialize
17   **     RSTSL_Finish
18   **
19   **     Internal Functions:
20   **
21   ** Compile-Time Options:
22   **     This section must list any compile time definitions
23   **     which will affect this header.
24   **
     ***********************************************************************
     ******/

27   /* The following provides an RCS id in the binary that can be located
28   ** with the what(1) utility.  The intent is to keep this short.
29   */

31   #ifndef lint
32   static char RCS_id [] = "$RCSfile$ "
33                          "$Revision$ "
34                          "$Date$" ;
35   #endif

38   /*
39    * Feature test switches.
40    *
41    * Standard defines required to turn on OS features go here.
42    *
43    * The following is required for code that uses POSIX API's.
44    * Remove for non-POSIX, non-portable code.
     */
46   #define _POSIX_SOURCE 1

49   /*
50    * System headers.
51    */
52   #include <sys/param.h>
53   #include <dirent.h>
54   #include <dlfcn.h>

57   /*
58    * Epoch headers.
59    */
60   #include <eb/eb_port.h>
61   #include <eb/rb_log.h>
```

```
64   /*
65    * Local headers
66    */
67   #include <RSLinterns.h>
68   #include <RSLstartup.h>

71   /*
72    * #defines, structures, typedefs local to this source file
73    */

75   static errno_ty init_plugins( restore_context *rcp );
76   static int validate_plugin( struct pluginData *piDataPtr );

79   /*
80    * External declarations
81    */
82    * This is the global "restore context" that will be used
83    * throughout the rest of the restore operations.
84    */
85   struct restore_context *rcp = NULL;

88   /*
89    * Definitions of the names of the plugin functions in the piFuncArray
90    * of the pluginData structure.
91    *      These must be in the same order and position
92    * as the piFuncIndex values defined in RSLplugin.h.
93    */

94   char *piFuncNames[piFuncIndexLast+1] = {
95       "RSTPI_Identify",
96       "RSTPI_Initialize",
97       "RSTPI_GetTopLevelObjects",
98       "RSTPI_SetTopLevelObject",
99       "RSTPI_GetNextLevelObjects",
100      "RSTPI_ClearRestoreContext",
101      "RSTPI_Submit",
102      "RSTPI_GetTopLevelTemplates",
103      "RSTPI_DoesAlternateExist",
104      "RSTPI_MarkObject",
105      "RSTPI_UnmarkObject",
106      "RSTPI_IsObjectMarked",
107      "RSTPI_IsObjectMarkable",
108      "RSTPI_GetAllBackupTimes",
109      "RSTPI_GetCurrentBackupTime",
110      "RSTPI_SetBackupForTime",
111      "RSTPI_SetPrevBackup",
112      "RSTPI_SetNextBackup",
113      "RSTPI_SetFirstBackup",
114      "RSTPI_SetMostRecentBackup",
115      "RSTPI_IsTherePrevBackup",
116      "RSTPI_IsThereNextBackup",
117      "RSTPI_IsTherePrevBackupForTime",
118      "RSTPI_IsThereNextBackupForTime",
119      "RSTPI_Finish",
120      "RSTPI_StartRestore",
121      "RSTPI_FindRestorableObjects",
122      "RSTPI_FindResults",
123      "RSTPI_GetNecessaryMedia"
124  };
```

```
127  /*********************************************************************
128   *
129   * RSTSL_Initialize:
130   *
131   * This function takes care of all the initialization for a restore
132   * session. This must be called prior to any of the other functions
133   * in the Restore API.
134   *
135   * Parameters:
136   *
137   *     userName (I) - The name of the user.
138   *
140   *********************************************************************/
141  eerrno_ty
142  RSTSL_Initialize( const char *userName )
143  {
145      eerrno_ty status = E_SUCCESS;
146
147      /*
148       * If we have not yet allocated space for a restore_context
149       * structure, do so now. If we have already done so,    just clear it
150       * now.
151       */
152      if (NULL == rcp)
153      {
154          rcp = (struct restore_context *)malloc(sizeof(
155                          struct restore_context));
156          if (NULL == rcp)
157          {
158              rec_api_log_csm(SUB_CSM_NOMEM, NULL);
159              return(EP_RB_RECOVER_NOMEM);
160          }
162      }
165      memset(rcp, 0, sizeof(struct restore_context));
167      rcp->rc_human_uidname = esl_strdup( userName );
168
170      if (!rcp->rc_human_uidname) {
171          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
172          return(EP_RB_RECOVER_NOMEM);
173      }
174
175      /*
177       * Set the appropriate field in the recovery context to indicate
179       * that this recover session is based on the Recover API.
180       * This flag is in place for historical reasons but is used by
181       * other parts of the Recover API library.
183       */
184      rcp->gui_mode = 1;
185
186      /*
188       * Initialize the logging mechanism.
189       */
190      if (status = rbrlog_begin(rcp, progname))
         {
             return(status);
         }

         /*
          * Initialize the few "recover context" variables that we can at
          * this early stage.
```

```
191       */
193      setup_proc(rcp);
195      /*
196       * The following call will:
197       *   -Initialize the saveset datatbase.
198       *   -Infer any information we can at this point.
199       */
201      if (status = startup(rcp))
202      {
203          return(status);
204      }
206      /* Do plugins setup: Find and initialize all valid restore plugin
                                                               libs: */
208      status = init_plugins( rcp );
210      return( status );
211      /* End of RSTSL_Initialize() */
         }
```

```
214   /******************************************************
215    *
216    * RSTSL_Finish
217    *
218    * Function Description:
219    *
220    * This function terminates a restoral session,
                             but not while a restore is in
221    * progress.
             It will be rejected if a restore is currently being executed.
222    * This routine will clean up any local memory used in the session.
223    *
224    * Parameters:
225    *
226    * none
227    *
       */

229    eerno_ty
230    RSTSL_Finish( void )
231 1  {
232 1     int mc_n;

235 1     eerno_ty err = E_SUCCESS;

237 1     if (NULL == rcp)
238 2     {
239 2        return( E_SUCCESS );
240 1     }
241 1     RemoveSubmitFiles();
242 1     /*
243 1      * Call rbr_cleanup() which kills the aux proc(
                                          s), unlocks the work
244 1      * item, then calls rbrlog_end(
245 1      * the log file.
246 1      */                 ) to enter the last logs and to close

248 1     rbr_cleanup(rcp);

250 1     /*
251 1      * Deallocate the memory of restore_context and the related
252 1      * structures.
253 1      */
255 1     if (NULL != rcp->rc_mcp)              /* Free the multicat structures */
256 2     {
257 2        mcat_destroy(rcp->rc_mcp);
258 1     }

260 1     /*
261 1      * Free the mark bit map space
262 1      */
264 1     for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
265 2     {
266 2        if (NULL != rcp->rc_marks[mc_n])
267 3        {
268 3           free(rcp->rc_marks[mc_n]);
269 2           rcp->rc_marks[mc_n] = NULL;
270 2        }
271 1     }
273 1     if (NULL != rcp->rc_nmarks_by_plane)
```

```
274 2     {
275 2        free(rcp->rc_nmarks_by_plane);
276 1     }

278 1     /*
279 1      * Free the configuration structures
280 1      */

282 1  #if 0
283 1     if (NULL != rcp->rc_cfgname)
284 2     {
285 2        free(rcp->rc_cfgname);
286 1     }
287 1  #endif

289 1     if (NULL != rcp->rc_config)
290 2     {
291 2        rbc_freeconfig(rcp->rc_config);
292 1     }

294 1     /*
295 1      * Free the DS_NONE structures array
296 1      * Note that even though rc_dsnones is the head of linked list
297 1      * of dsnone_info structures, the list is allocated via malloc
298 1      * as an array initially (ref. alloc_plane_arrays()), therefore
299 1      * we can do a free here.
300 1      */

302 1     if (NULL != rcp->rc_dsnones)
303 2     {
304 2        free(rcp->rc_dsnones);
305 1     }

307 1     /*
308 1      * Free the volume list structures.
309 1      */

311 1     if (NULL != rcp->ebvllist)
312 2     {
313 2        (void)ebvl_volidlist_destructor(
                            rcp->ebvllist, EBVL_DESTROY_ALL);
314 1     }

316 1     /*
317 1      * Free the plugin related data
318 1      */

320 1     rcp->rc_backup_app = 0;
321 1     while (rcp->currentPiptr = rcp->pilist)
322 2     {
323 2        rcp->rc_backup_app++;
324 2        rcp->appData = rcp->currentPiptr->appData;
325 2        /* allow plugin to clean up and close .so: */
326 2        if ( E_SUCCESS != (err =
327 2           rcp-> currentPiptr-> piFuncArray[ PIFuncIndexFinish ] (
                     rcp) ) )
328 3        {
329 3           /* log error, continue */
330 3           rbe_user_error( err,
331 3              "RSTPL_Finish failed for restore plug-in
                        library %s\n",
332 3              ((struct pluginIDdata *)(
                        rcp-> currentPiptr-> idData)) -> name ) ;
333 3        }
334 2        dlclose( rcp-> currentPiptr-> libHdl ) ;
335 2        rcp->pilist = rcp->pilist->next;
```

```
336  2        free (rcp->currentPIptr);
337  1    }

339  1    /*
340  1     * Free the various simple string buffers
341  1     */

343  1    if (NULL != rcp->rc_top_level_object_name)
344  2    {
345  2        free(rcp->rc_top_level_object_name);
346  1    }

348  1    if (NULL != rcp->rc_template_name)
349  2    {
350  2        free(rcp->rc_template_name);
351  1    }

353  1    if (NULL != rcp->rc_workitem_name)
354  2    {
355  2        free(rcp->rc_workitem_name);
356  1    }

358  1    if (NULL != rcp->rc_human_uidname)
359  2    {
360  2        free(rcp->rc_human_uidname);
361  1    }

363  1    if (NULL != rcp->rc_effective_uidname)
364  2    {
365  2        /* don't free, its internal: free(rcp->rc_effective_uidname) */;
366  1    }

368  1    if (NULL != rcp->rc_client_rbuname)
369  2    {
370  2        free(rcp->rc_client_rbuname);
371  1    }

373  1    if (NULL != rcp->rc_client_hostname)
374  2    {
375  2        free(rcp->rc_client_hostname);
376  1    }

378  1    if (NULL != rcp->rc_client_scriptname)
379  2    {
380  2        /* don't free, its internal: free(rcp->rc_client_scriptname);
         */
381  1    }

383  1    if (NULL != rcp->rc_client_dirtop)
384  2    {
385  2        free(rcp->rc_client_dirtop);
386  1    }

388  1    if (NULL != rcp->rc_cmd_context)
389  2    {
390  2        /* don't free -- its internal/temp data: free(
391  1            rcp->rc_cmd_context); */

393  1    if (NULL != rcp->rc_source_client_hostname)
394  2    {
395  2        free(rcp->rc_source_client_hostname);
396  1    }

398  1    if (NULL != rcp->rc_cpiogen_executable)
```

```
399  2    {
400  2        /* don't free, its internal: free(rcp->rc_cpiogen_executable);
         */
401  1    }

403  1    if (NULL != rcp->rc_plugin_wi_types)
404  2    {
405  2        free(rcp->rc_plugin_wi_types);
406  1    }

408  1    if (NULL != rcp->rc_pwd)
409  2    {
410  2        free(rcp->rc_pwd);
411  1    }

413  1    /*
414  1     * Finally, deallocate the restore_context itself
415  1     */

417  1    free(rcp);
418  1    rcp = NULL;

422  1    return( err );
423  1    }    /* RSTSL_Finish */
```

```
427  /**********************************************************
428   *
429   * init_plugins
430   *
431   *   Function Description:
432   *      This function locates, opens,
                validates and initializes all restore
433   *      plug-in (shared) libraries.  They must be located in
434   *      /usr/epoch/EB/cure_plugin (
                eb_cure_plugin_dir).  All .so files in that
435   *      directory are opened and validates for version# and presence of
                all
436   *      mandatory functions.
                The RSTPI_Identify function is called for each
437   *      library to determine which optional features are supported,
                and that
438   *      the corresponding functions are present.  Finally,
                the RSTPI_Initialize
439   *      function is called for each valid library.
440   *
441   *   Parameters:
442   *
443   *   Inputs:
444   *      rcp        (I)    - Pointer to restore context
445   *
446   *   Outputs:
447   *      none
448   *
449   *   Returns:
450   *      E_SUCCESS or EP_RB_RECOVER_xxx
451   *
452   *   Logic/pseudo code:
453   *
454   *      open plugin dir
455   *      while read_next_entry succeeds
456   *         verify .so file (else continue)
457   *         open shared library file (else continue)
458   *         on errors below:
459   *            close shared library file
460   *            continue
461   *         fetch all mandatory function addresses
462   *         call Identify function
463   *         validate version number
464   *         fetch all indicated optional function addrs
465   *         call Initialize function
466   *         add workitem types to composite exclusion list
467   *         add to valid plugin list
468   *      close plugin dir
469   *
470   */
471  static eerrno_ty init_plugins( restore_context *rcp )
472  {
473     DIR                 *dirp;
474     struct dirent       *direntp;
475     eerrno_ty           status = E_SUCCESS;
476     struct pluginData   *piDataPtr = NULL;
477     struct pluginData   *piListPtr = NULL;
478     int                 val_result;
479     struct pluginIDdata *idDataPtr;
480     char                *tmp_types;
481     int                 shlib_dirlen;
482     char                shlib_path [MAXPATHLEN];
```

```
484     /* open plugin directory or bust */
485     if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
486     {
487        rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
488 #if 1
489        return E_SUCCESS;          /* allow continuation w/o plugins */
490 #else
491        return EP_RB_RECOVER_NO_PLUGINS;          /* later do this */
492
493 #endif
494     }

495     strcpy( shlib_path, eb_cure_plugin_dir );
496     strcat( shlib_path, "/" );
497     shlib_dirlen = strlen (shlib_path);

499     /* loop thru entries in directory*/
500     while (NULL != (direntp = readdir( dirp )))
501     {
502        /* allocate next plugin data structure */
503        if (NULL == piDataPtr)
504        {
505           if (NULL == (piDataPtr
506                       = calloc( 1, sizeof(
507                         struct pluginData) )))
508           {
509              status = EP_RB_RECOVER_NOMEM;
510              break;              /* fall thru to cleanup */
511           }
512        }

513        if (NULL == strstr( direntp->d_name, ".so" ) )
515           continue;              /* skip this guy */

517        strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
518        if (NULL == (piDataPtr->libHdl
519                    = dlopen( shlib_path, RTLD_NOW )))
520        {
521           rbe_user_error( 0,
522              "Error opening restore plug-in library
                  %s: %s\n",
524              direntp->d_name, dlerror() );
525           continue;              /* skip this one */
526        }

528        /* Fetch addresses of all mandatory functions and */
529        /* Do Identify processing: call it, save options, validate */

531        if (0 != (val_result = validate_plugin(
532                               piDataPtr )))
533        {
534           if (val_result == -1 || val_result == -4)
535           {
536              rbe_user_error( 0,
537                 "Functions missing from restore plug-in library %s:
                     %s\n",
539                 direntp->d_name, dlerror() );
540           }
538           else if (val_result < 0)
537           {
536              rbe_user_error( 0,
541                 "Validation failed for restore plug-in
                     library %s\n",
540                 direntp->d_name );
541           }
```

```
542 3      else
543 4      {
544 4          rbe_user_error( val_result,
545 4      "RSTPI_Identify failed for restore plug-in library
546 4                              %s\n",
547 3                              direntp->d_name );
549 3      }
550 3      dlclose( piDataPtr->libHdl );
551 3      piDataPtr->libHdl = NULL;
552 2      continue;              /* close .so on errors */
                                  /* on any error, skip this lib */
554 2      }
555 2      /* let DC plug-in do its initialization */
555 2      rcp->appData = NULL;
                                  /* enter plugin with clean appdata */
556 2      status =
557 2      piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
558 3      if (E_SUCCESS != status)
559 3      {
560 3          rbe_user_error( status,
          "RSTPI_Initialize failed for restore plug-in library
                                  %s\n",
561 3                              direntp->d_name );
562 3      dlclose( piDataPtr->libHdl );
563 3          piDataPtr->libHdl = NULL;
                                  /* close .so on errors */
564 3      status = E_SUCCESS;
                                  /* this wasn't fatal */
565 3      continue;
                                  /* on any error, skip this lib */
566 2      }

568 2      /* add piDataPtr to valid plugin list */
569 2      if (NULL == pilistPtr)
570 2          rcp->pilist= piDataPtr;
                                      /* first in list */
572 2      else
573 2          pilistPtr->next = piDataPtr;
574 2                                /* link from prev */
575 2      pilistPtr = piDataPtr;
576 2                                /* new end of list */
577 2      piDataPtr = NULL;

578 2      /* save plugin's appData */
580 2      piDataPtr->appData = rcp->appData;
581 2      rcp->appData = NULL;

582 2      /* add workitem types to composite exclusion list */
583 3      idDataPtr = (struct pluginIDdata *)pilistPtr->idData;
584 3      if (idDataPtr->num_types > 0)
585 3      {
586 4          tmp_types = calloc( 1, 1 + idDataPtr->num_types
587 4                              + rcp->rc_num_plugin_wi_types)
588 4          if (NULL == tmp_types) {
589 4              status = EP_RB_RECOVER_NOMEM;
590 3              break;
591 4          }
592 3          if (NULL != rcp->rc_plugin_wi_types)
593 4          {
594 4              /* move old list to new buffer and free old list */
                   memcpy( tmp_types,
                           rcp->rc_plugin_wi_types,
                           rcp->rc_num_plugin_wi_types );
```

```
595 4          free( rcp->rc_plugin_wi_types );
596 4          }
597 3      memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
598 3              idDataPtr->wi_types,
599 3              idDataPtr->num_types );
600 3      rcp->rc_num_plugin_wi_types +=
601 3                              idDataPtr->num_types;
602 3      tmp_types[rcp->rc_num_plugin_wi_types] = 0;
603 2      rcp->rc_plugin_wi_types = tmp_types;
604 1      }
          }

606 1  (void)closedir( dirp );

608 1  /* free up leftovers: */
609 1  if (NULL != piDataPtr)
610 1      free (piDataPtr);

612 1  if (E_SUCCESS != status)
613 2  {  /* Free contents of plugin list: */
614 2      while (NULL != (piDataPtr = pilistPtr))
615 3      {  /* allow plugin to clean up and close .so: */
616 3          rcp->appData = piDataPtr->appData;
617 3          piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                       rcp );
618 3          dlclose( piDataPtr->libHdl );
619 3          pilistPtr = piDataPtr->next;
620 3          free (piDataPtr);
621 2      }
622 1  }

624 1  return status;
625    }
```

```
626            /* init_plugins */

628    /****************************************************************
629     * validate_plugin
630     *
631     *     Function Description:
632     *
633     *     This function retrieves the addresses of the mandatory plugin
634     *     and stores them in the function pointer array.    If any function is missing
635     *     it returns -1.
636     *     It then calls the identify function and verifies whe plugin
637     *     version,
638     *     and finds its optional functions. Specific error values are
639     *     returned on version mismatch and missing optional functions.
640     *
641     *     Parameters:
642     *     Inputs:
643     *         piDataPtr (
644     *         I)    - Pointer to plugin data structure with libHdl set
645     *     Outputs:
646     *         piFuncArray in piDataPtr is loaded with pointers to plugin
                       functions
647     *     Returns:
648     *         0 on success
649     *        -1 on any missing required functions
650     *        -2 if version validation fails OR identify returns junk
651     *        -3 if workitem type validation fails
652     *        -4 on any missing optional functions indicated by options
                       flags
653     *        +n (
654     *         EB_RB_RECOVER_xxx) for error codes returned from Identify function
655     ****************************************************************/

657  1  static int validate_plugin( struct pluginData *piDataPtr )
658  1  {
659  1      int                     index;
660  1      errno_ty                status;
661  1      struct pluginIDdata     *idData;

663  1      for( index = 0; index <= PIFuncIndexLastBasic; index++ )
664  2      {
665  2          if (NULL == (piDataPtr->piFuncArray[index]
666  2                  = (piFuncPtr) dlsym( piDataPtr->libHdl,
667  2                          piFuncNames[index]
668  2          )))
669  2          {
670  1              return -1;
                }

672  1      /* call identify and validate: */
673  1      status = piDataPtr->piFuncArray[PIFuncIndexIdentify](
674  1                      &piDataPtr->idData );
675  1      if (status != E_SUCCESS)
676  1          return status;
677  1      if (NULL == (idData = (
678  1          struct pluginIDdata *)piDataPtr->idData)
                return -2;
```

```
680  1      if (idDataPtr->version != RSTPI_VERSION)
681  2      {  /* only version 1 supported so far */
682  2          piDataPtr->idData = NULL;
683  2          return -2;
684  1      }

686  1      if (idDataPtr->num_types && !idDataPtr->wi_types)
687  2      {  /* count cant be positive with null pointer */
688  2          piDataPtr->idData = NULL;
689  2          return -3;
690  1      }

692  1      /* if startRestore option set, get its addr or bust */
693  1      if ( ( (RSTPI_OPTION_SPECIAL_START
694  1              == (idDataPtr->options & RSTPI_OPTION_MASK_START))
695  1          && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexStartRestore]
696  1                  = (piFuncPtr) dlsym( piDataPtr->libHdl,
697  1                          piFuncNames[PIFuncIndexStartRestore] )))

699  1          ||
700  1          (NULL == (
701  1              piDataPtr->piFuncArray[PIFuncIndexFindResults]
702  1              = (piFuncPtr) dlsym( piDataPtr->libHdl,
703  1                      piFuncNames[PIFuncIndexFindResults])
704  1          )   )   )

705  1          || (NULL == (
706  2              piDataPtr->piFuncArray[PIFuncIndexFind]
707  1              = (piFuncPtr) dlsym( piDataPtr->libHdl,
708  1                      piFuncNames[PIFuncIndexFind])

709  1          )   )   )

710  1      /* OR if special find option set, get its addr or bust */
711  1      || ( (RSTPI_OPTION_SPECIAL_FIND
712  1              == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))

713  1      /* OR if special getMedia option set, get its addr or bust */
714  1      || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
715  1              == (
716  2                  idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
717  2          && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
718  2                  = (piFuncPtr) dlsym( piDataPtr->libHdl,
719  1                          piFuncNames[PIFuncIndexGetMedia] )))

721  1      ) ) )
            {
722  1          piDataPtr->idData = NULL;
                return -4;
            }

            return 0;
        }
```

723

```
/*    validate_plugin  */
```

```
  2  /******************************************************************
  3  **
  4  ** File Name:   RSLplugin.h
  5  **
  6  **
  7  ** Copyright (c) 1999 by EMC Corporation.
  8  **
  9  ** Purpose:
 10  **      Two-fold:
 11  **      - Define prototypes of functions in a restore plugin
 12  **        library
 13  **      - Define prototypes of common functions exported from
 14  **        the restore engine for use by restore plugin libraries.
 15  **        These functions are part of the restore_legacy library.
 16  **
 17  **      This header defines the API that a backup application must
 18  **      supply to be compatible with the new Restore API.
 19  **
 20  **      The Restore API provides client-server based restore
 21  **      functionality formerly provided by the RECOVER_API. The
 22  **      plug-in API is provided in a shared library, designed to
 23  **      be called by the Restore Service Library -- the generic
 24  **      server side component of the Restore API that executes as
 25  **      part of the Restore Engine (server).
 26  **
 27  **      Compile-Time Options:
 28  **          Need:
 29  **          Need header files:
 30  **              #define _POSIX_SOURCE 1
 31  **              <ebutil/ebutil.h>
 32  **              <restore/restore_api.h>
 33  **              <restore/restoreRPC.h>
 34  ******************************************************************/

 33  #ifndef H_RSLPIAPI
 34  #define H_RSLPIAPI

 36  #ifdef  _cplusplus
 37     extern "C" {
 38  #endif

 40  #include <ebutil/ebutil.h>
 41  #include <eerrno/e_eb.h>
 42  #include <restore/restore_api.h>
 43  #include <restore/restoreRPC.h>

 46  /* The following definition identifies the version of the Restore
 47   * Plugin header that a plug-in library was built with. Whenever a
 48   * plugin function prototype changes, or a plugin function is added,
 49   * this value should be updated. It will be used by the restore
 50   * engine code to verify compatibility with individual restore
 51   * plugin libraries.
     */
```

```
 53  #define RSTPI_VERSION           1

 56  /* Definitions of the indices of the plugin function addrs in the piFuncArray
 57   * of the pluginData structure.
 58   */

 60  /* Mandatory functions: */
 61  #define PIFuncIndexIdentify      0
 62  #define PIFuncIndexInitialize    1
 63  #define PIFuncIndexSetTLO        2
 64  #define PIFuncIndexGetTLO        3
 65  #define PIFuncIndexSetNLO        4
 66  #define PIFuncIndexGetNLO        5
 67  #define PIFuncIndexClearRC       6
 68  #define PIFuncIndexSubmit        7
 69  #define PIFuncIndexGetTLTemps    8
 70  #define PIFuncIndexDoesAltExst   9
 71  #define PIFuncIndexMark          10
 72  #define PIFuncIndexUnmark        11
 73  #define PIFuncIndexIsObMarked    12
 74  #define PIFuncIndexIsObMarkable  13
 75  #define PIFuncIndexGetAllTimes   14
 76  #define PIFuncIndexGetCurTime    15
 77  #define PIFuncIndexSetBkupTime   16
 78  #define PIFuncIndexSetPrevTime   17
 79  #define PIFuncIndexSetNextTime   18
 80  #define PIFuncIndexSetFirstTime  19
 81  #define PIFuncIndexSetMostRcnt   20
 82  #define PIFuncIndexIsPrevBkup    21
 83  #define PIFuncIndexIsNextBkup    22
 84  #define PIFuncIndexIsPrvForTime  23
 85  #define PIFuncIndexIsNxtForTime  24
 86  #define PIFuncIndexLastBasic     24

 88  /* Optional functions: */
 89  #define PIFuncIndexStartRestore  25
 90  #define PIFuncIndexFind          26
 91  #define PIFuncIndexFindResults   27
 92  #define PIFuncIndexGetMedia      28
 93  #define PIFuncIndexLast          28

 95  /* Definitions of the names of the plugin functions in the piFuncArray
 96   * of the pluginData structure.        These must be in the same order and position
 97   * as the PIFuncIndex values above, and are initialized in RSLinitfin.c.
 98   */

100  char *piFuncNames[PIFuncIndexLast+1];

103  typedef eerrno_ty (*piFuncPtr)();              /* generic plugin func prototype */

105  typedef struct restore_context restore_context;     /* for function prototypes */

108  /* callback function prototypes: */

110  typedef boolean_ty   (*RSTPI_MarkProgressProc)(
                                                   unsigned long totalMarks);
111  typedef boolean_ty   (*RSTPI_SubmitProgressProc)(
                                                   unsigned long totalElements);
```

```
112  1   typedef boolean_ty    (*RSTPI_FindProgressProc)(
                                unsigned long progressCount);
113  1   typedef boolean_ty    (*RSTPI_QuitTest)( void );

116  1   /** Definitions of the data that the RSTPI_Identify returns. **/

118  2   struct pluginIData {
119  2     u_long version;
                                /* Version of the plugin header that the
                                   library was built with */
120  2     char *name;
121  2                          /* Name of the backup application (64 byte
                                   buffer address) */
122  2     u_long options;
                                /* Bit mask identifying the optional plug-in
123  2                             features supported.
124  2                             The bit definitions for
125  2                             this parameter (
126  2                             RSTPI_OPTION_MASK...) are
127  2                             defined below. */
           char *wi_types;
                                /* pointer to array of workitem types supported
128  2                             by the plugin */
129  2     u_short num_types;
                                /* Number of witypes in the witypes array */
130  1   };

132  1   /** Definitions of the options bits in the RSTPI_Identify function
                                output: **/
133  1   /* NOTE: The output structure containing this data is defined in
                                RSLcontext.h */

135  1   /* This bit indicates whether the plug-in supplies its own function to
         * execute the actual restore.
         */
136  1   #define RSTPI_OPTION_MASK_START          0x1
137  1
138  1   #define RSTPI_OPTION_SPECIAL_START       0x1
139  1
140  1   #define RSTPI_OPTION_GENERIC_START       0x0

142  1   /* The following bits indicate whether and how the plug-in supports
143  1   * function for searching its catalogs for specific objects.      the find
144  1   * indicates if find is supported,                                The first bit
                                the second bit indicates whether the plugin
145  1   * supplies its own find function,
                                or that the generic find function that
146  1   * searches the mcat data can be used.
147  1   */
148  1   #define RSTPI_OPTION_MASK_FIND           0x6
149  1   #define RSTPI_OPTION_FIND_SUPPORTED      0x4
150  1   #define RSTPI_OPTION_SPECIAL_FIND        0x6
151  1   #define RSTPI_OPTION_GENERIC_FIND        0x4
152  1   #define RSTPI_OPTION_NO_FIND             0x0

154  1   /* The following flag indicates whether objects can be restored from
155  1   * backups (
                                backup planes) in the same restore. If not, the user will be
156  1   * warned that changing backup planes will cause objects marked for
                                restore
157  1   * from a different backup time to be automatically unmarked.
         */
158  1
159  1   #define RSTPI_OPTION_MASK_MULTI_PLANE         0x8
160  1   #define RSTPI_OPTION_MULTI_PLANE_SUPPORTED    0x8
```

```
162  1   /* The following flag indicates whether the plugin provides an
163  1   * RSTPI_GetNecessaryMedia function to return the list of media needed
                                to
164  1   * restore the restorable objects currently marked.
                                Otherwise the plugin must
165  1   * maintain the ebvl_volidlist_ty * list in the restore_context
                                whenever
166  1   * objects are marked and unmarked.
167  1   */
168  1   #define RSTPI_OPTION_MASK_GET_MEDIA        0x10
169  1   #define RSTPI_OPTION_SPECIAL_GET_MEDIA     0x10
170  1   #define RSTPI_OPTION_GENERIC_GET_MEDIA     0x00

172  1   /* The following flag indicates whether objects can be restored through
173  1   * a symmetrix ('SCSI')
174  1   */
175  1   #define RSTPI_OPTION_MASK_SYMM_RESTORE     0x20
176  1   #define RSTPI_OPTION_SYMM_RESTORE_ALLOWED  0x20

178  1   /*****************************************************************
179  1   *                 Plug-in API function definitions              *
180  1   *****************************************************************/

183  1   /*****************************************************************
184  1   * Identify:
185  1   *
186  1   * This function is called once,
                                to identify and validate the plug-in with
187  1   * regard to the operating restore engine.
                                The version number is checked
188  1   * for compatibility with the restore engine,
                                and the optional features
189  1   * of the plug-in are specified.
190  1   *
191  1   * Parameters:
192  1   * pi_defs (
193  1   *     O)  - address of the structure identifying the plugin to the
194  1   *           restore engine. Its fields consist of:
195  1   * name      - Name of the backup application (
                                64 byte buffer address)
196  1   * options   - Bit mask identifying the optional plug-in features
                                supported
197  1   *             The bit definitions for this parameter (
                                RSTPI_OPTION_MASK...) are
198  1   *             defined above.
199  1   * wi_types  - pointer to array of workitem types supported by the
                                plugin
200  1   * num_types - number of witypes in the witypes array
201  1   *
202  1   * Returns:
203  1   * E_SUCCESS          on success
204  1   * EP_RB_RECOVER_xxx  on error
205  1   *
206  1   *****************************************************************/
207  1   eerrno_ty RSTPI_Identify( const struct pluginIData **pi_defs );

210  1   /*****************************************************************
211  1   * Initialize:
```

```
212 1  *  This function is called once, to allow the plug-in to perform its
213 1  *  'local' initialization.
214 1  *
215 1  *      The restore context will have already been
216 1  *      initialized with the config file parsed,
217 1  *      and the human user fields
218 1  *      set.
219 1  *
220 1  *  Parameters:
221 1  *      context (I) - Pointer to the restore context
222 1  *
223 1  *  Returns:
224 1  *      E_SUCCESS           on success
225 1  *      EP_RB_RECOVER_xxx   on error
226 1  *
       **********************************************************/
       eerrno_ty RSTPI_Initialize( restore_context *context );

229 1  /**********************************************************
230 1   *  Finish:
231 1   *
232 1   *  This function is called to allow the plug-in to clean up its internal
233 1   *  storage when a restore session is ending.
234 1   *
235 1   *  Parameters:
236 1   *      context (I) - Pointer to the restore context
237 1   *
238 1   *  Returns:
239 1   *      E_SUCCESS           on success
240 1   *      EP_RB_RECOVER_xxx   on error
241 1   *
242 1   *
243 1   **********************************************************/
       eerrno_ty RSTPI_Finish( restore_context *context );

247    /**********************************************************
248 1   *  Get Top Level Objects
249 1   *
250 1   *  This function is called to retrieve the configurable backup objects (work
251 1   *  items for network backups and work item sets for Symmetrix Connect backups,
252 1   *  which are restorable for the given client.
253 1   *
254 1   *  It is a GOAL of this routine to return all objects ever backed
255 1   *  up successfully.  For network backups, though, it only looks in the config
256 1   *  file for 'top level objects' of the given client.
257 1   *
258 1   *  While the restore API will be called repeatedly to retrieve a maximum
259 1   *  number of items on each call, this plug-in call must retrive the whole
260 1   *  set of applicable backup objects.  The generic restore service library
261 1   *  will manage the composite list of top level objects from all backup apps.
262 1   *
263 1   *  Parameters:
264 1   *      context      (I) - Pointer to the restore context
265 1   *      sourceHost   (I) - the name of the host whose backups are being restored
266 1   *      topLevObjs   (O) - ptr to linked list of Top Level Objects
```

```
267 1   *      numberEntries ( O) - the real number of objects returned in the list
268 1   *
269 1   *  Returns:
270 1   *      E_SUCCESS           on success
271 1   *      EP_RB_RECOVER_xxx   on error
272 1   *
273 1   **********************************************************/
       eerrno_ty
       RSTPI_GetTopLevelObjects( restore_context          *context,
                                 const char               *sourceHost,
                                 struct RSTRPC_tlo_list   **topLevObjs,
                                 short                    *numberEntries );

280 1  /**********************************************************
281 1   *  Set Toplevel Object:
282 1   *
283 1   *  This function is called to notify the plug-in that one of its top
284 1   *  level objects has been selected (for browsing and marking). The plug-in
285 1   *  should perform whatever validation and initialization is needed to prepare
286 1   *  for browsing and marking this top level object.
287 1   *  If this call returns success,      then RSTPI_GetNextLevelObjects will be
288 1   *  called to retrieve the highest level restorable objects for this
289 1   *  backup object.
290 1   *
291 1   *  NOTE: This function is responsible for setting the template_name and
292 1   *  saveset_thread elements of the restore context.
293 1   *
294 1   *  Returns:
295 1   *      E_SUCCESS           on success
296 1   *      EP_RB_RECOVER_xxx   on error
297 1   *
298 1   **********************************************************/
299 1  eerrno_ty
300 1  RSTPI_SetToplevelObject( restore_context             *context,
301 1                           struct RSTRPC_top_level_obj *backupObject );
302 1
303 1  /**********************************************************
304 1   *  Get Next Level Objects:
305 1   *
306 1   *  This function is intended to allow retrieval of the children
       *  of a given parent object.
308 1   *      The caller specifies the parent object and
309 1   *  whether or not to include bad files. Even though the objects are
310 1   *  returned in a linked list, there could conceivably be thousands of
311 1   *  child objects, so the caller must specify the maximum number
312 1   *  of children to return. The caller is returned a token (cookie) to allow
313 1   *  continuing on the next call to thid function.
314 1   *
315 1   *  Parameters:
316 1   *      context
317 1   *      parentObject
318 1   *      objectLevel
319 1   *                  (I) - Pointer to the restore context
320 1   *                  (I) - the parent object
321 1   *                  (I) - specifies whether parentObject is a top level or
322 1   *                        container object
323 1   *
```

```
324  1   *     objects       O)   - a pointer to receive the start of the objects list
325  1   *     cookie        (IO) - a place holder for the list position
326  1   *     maxEntries    (I)  - the maximum number of objects to return
327  1   *     numberEntries (O)  - the real number of objects returned in the list
328  1   *     allowBadFiles (I)  - flag whether or not to include bad files
329  1   *
330  1   *  Returns:
331  1   *     E_SUCCESS          on success
332  1   *     EP_RB_RECOVER_xxx  on error
333  1   *
334  1   ****************************************************************/
336  1   eerrno_ty
337  1   RSTPI_GetNextLevelObjects( restore_context         *context,
339  1                              restorableObjectPtr     parentObj,
340  1                              enum RSTRPC_ObjectLevel objectlevel,
341  1                              struct RSTRPC_uro_list  **objects,
342  1                              long                    *cookie,
343  1                              long                    maxEntries,
344  1                              long                    *numberEntries,
         1                          const boolean_ty        allowBadFiles );
346  1   /****************************************************************
347  1    *
348  1    *  Clear Restore Context :
349  1    *
350  1    *  This function is called to notify the plug-in that its currently
                selected
351  1    *  top level object is no longer selected ( for browsing and marking). The
352  1    *  plug-in should perform whatever cleanup and memory deallocation is
353  1    *  appropriate.
354  1    *
355  1    *  Returns:
356  1    *     E_SUCCESS          on success
357  1    *     EP_RB_RECOVER_xxx  on error
358  1    *
359  1    *  Parameters:
360  1    *     context    (I)    - Pointer to the restore context
361  1    *
362  1    ****************************************************************/
363  1   eerrno_ty
         1   RSTPI_ClearRestoreContext( restore_context *context );
365  1   /****************************************************************
366  1    *  Submit
367  1    *
368  1    *  This function creates a submit object from the currently marked
369  1    *  restorable objects.
370  1    *     The ID of the created submit object is passed to
371  1    *  EDMRST_Start to begin execution of the restore.
372  1    *
373  1    *  Parameters:
374  1    *     context    (I)   - Pointer to the restore context
375  1    *     hostName   (I)   - host to restore to (only if inPlace == False)
376  1    *     policy     (I)   - The overwrite policy to use
377  1    *     inPlace    (I)   - Flag if the restoral is to be in original locations
378  1    *     directory  (I)   - directory to restore to (
379  1    *                          only if inPlace == False)
380  1    *     transport  (I)   - Indicator of transport the restoral is to be over (SCSI
                                     or network)
```

```
381  1    *     submitObjIDptr (IO) - ID of the submit user object created to describe
382  1    *                            the restore
383  1    *     progressCB    (I)  - pointer to callback function to report progress and
384  1    *                            test for cancellation
385  1    *
386  1    ****************************************************************/
387  1   eerrno_ty RSTPI_Submit( restore_context          *context,
388  1                           const char               *hostName,
389  1                           const OverwritePolicy    policy,
390  1                           const boolean_ty         inPlace,
391  1                           const char               *directory,
392  1                           const RestoreTransport   transport,
393  1                           int                      *submitObjIDptr,
394  1                           RSTPI_SubmitProgressProc progressCB );
396  1   /****************************************************************
397  1    *  StartRestore
398  1    *
399  1    *  This optional function begins the application-specific execution of
400  1    *  restoral of the objects in a submit object. The quittest callback
401  1    *  function must be called periodically to check for cancellation of
402  1    *     the restore.
403  1    *
404  1    *  Parameters:
405  1    *     context      (I)  - Pointer to the restore context
406  1    *     submitObjectID (I) - ID of the submit object which describes the restore
407  1    *     quitTestCB   (I)  - function to call to check for the quit signal
408  1    *
409  1    ****************************************************************/
410  1   eerrno_ty RSTPI_StartRestore( restore_context  *context,
411  1                                 int              submitObjectID,
412  1                                 RSTPI_Quittest   quitTestCB );
415  1   /****************************************************************
416  1    *  Find Routines:
417  1    *
418  1    *  RSTPI_FindRestorableObjects and RSTPI_GetFindResults
419  1    *
420  1    *  These two functions allow the user to find restorable objects. Returned
421  1    *  is a linked list of found objects. Each entry in the list contains a
422  1    *  pointer to a restorable object, and the backup time associated with the
423  1    *  object.
424  1    *
425  1    *  The find operation is performed by the asynchronous thread of the Restore
426  1    *  engine, with the RSTSL_FindRestorableObjects function calling
427  1    *  RSTPI_FindRestorableObjects if a plugin has its own find logic.
428  1    *  RSTPI_GetFindResults, is used to test for completion of the find, A second RPC call,
429  1    *  signal cancelation. A callback function pointer is passed into and to
430  1    *  RSTPI_FindRestorableObjects to allow testing for cancellation of
431  1    *  and to pass back progress information ( the find,
432  1    *  The second service library function, RSTPI_GetFindResults, is also used to
```

```
* retrieve the output of the find operation.
*
* RSTPI_FindRestorableObjects Parameters:
*
* context       (I) - Pointer to the restore context
* searchCriteria (I) - The criteria used for the search
* intr_callback (I) - Function to check for cancel and return progress
*
****************************************************
eerrno_ty
RSTPI_FindRestorableObjects ( restore_context         *context,
                              EBREC_SearchCriteriaRec *searchCriteria,
                              RSTPI_FindProgressProc   intr_callback
                            );

****************************************************
* RSTPI_GetFindResults Parameters:
*
* context        (I)  - Pointer to the restore context
* cancel         (I)  - requests cancellation of the find (if TRUE)
* maxEntries     (I)  - the maximum number of objects to return
* foundObjs      (O)  - a pointer to a linked list of found objects
* numberEntries  (O)  - the real number of objects returned in the list
* cookie         (IO) - a place holder for excess found objects
*
****************************************************/
eerrno_ty RSTPI_GetFindResults ( restore_context    *context,
                                 const boolean_ty    cancel,
                                 const long          maxEntries,
                                 struct RSTRPC_found_obj_list
                                                   **foundObjects,
                                 long               *numberEntries,
                                 long               *cookie );

****************************************************
* Get Top Level Templates:
*
* This function is required to retrieve the templates with which a
* backup object could have been backed up.
*
* Parameters:
* context        (I) - Pointer to the restore context
* topLevelObj    (I) - the top level object
* templates      (O) - pointer to the start of the list of templates
* numberEntries  (O) - the real number of templates returned in the list
*
****************************************************/
eerrno_ty
RSTPI_GetTopLevelTemplates ( restore_context           *context,
                             struct RSTRPC_top_level_obj *topLevelObj,
                             struct RSTRPC_name_list    **templates,
                             short                      *numberEntries );

/****************************************************
* Does Alternate Exist
*
```

```
* This routine determines if an alternate trailset exists for the
* given template.
*
* Parameters:
* context       (I) - Pointer to the restore context
* templateName  (I) - The name of the template to look for
* exists        (O) - Return flag for whether or not the alternate exists
*
****************************************************/
eerrno_ty
RSTPI_DoesAlternateExist ( restore_context       *context,
                           const template_name_ty templateName,
                           boolean_ty            *exists );

/****************************************************
* Mark Object
*
* The MarkObject operation takes a restorableObject and marks it, and
* possibly its descendant files for restoral based on the input criteria.
*
* Since the RSTPI_MarkObject call is an asynchronously executed operation
*
* in the Restore Engine that performs the marking, this function must
*
* periodically check for user-signalled cancelation, and update progress
*
* data using the progress callback function argument.
*
* NOTE: This functions is responsible for keeping the volumes needed list
*
* (ebvllist) element of the restore context up to date.
*
* Parameters:
* context       (I) - Pointer to the restore context
* thisObject    (I) - The restoral object;
*                      can be a leaf object (e.g. a
*                      file), or a container object (
*                      e.g., a directory).
* allowBadfiles (I) - allows marking of files of state BADDATA.
* descend       (I) - Should mark operation descend to operate on the content
*                      of container objects.
* BadFilesCount (O) - returns the file count with BADDATA.
* PermDenyFilesCount (O) - returns the file count with permission denied.
* fileMarked    (O) - return the total files marked after this mark occurred.
* lenMarkedFiles (O) - return the length of files marked after this mark
* dirMarked     (O) - return the total directories marked after this mark
* otherMarked   (O) - return the total "other" files marked after this mark.
* progressCB    (I) - pointer to callback function to report progress and
*                      test for cancellation
*
****************************************************/
eerrno_ty
RSTPI_MarkObject ( restore_context *context,
```

```
540 1    struct RSTRPC_user_restorable_object *thisObject,
541 1    boolean_ty                           allowBadFiles,
542 1    boolean_ty                           descend,
543 1    unsigned long                        *BadFilesCount,
544 1    unsigned long                        *PermDenyFilesCount,
545 1    unsigned long                        *fileMarked,
546 1    u_hyper                              *lenMarkedFiles,
547 1    unsigned long                        *dirMarked,
548 1    unsigned long                        *otherMarked,
549 1    RSTPI_MarkProgressProc               progressCB );
551 1  /*****************************************************************
552 1   * UnmarkObject
553 1   *
554 1   * The UnmarkObject operation takes a restorableObject and unmarks
555 1   * possibly its descendant files for restoral based on the input
        *                                                       it, and criteria.
556 1   * Since the RSTPI_UnmarkObject call is an asynchronously executed
        *                                                       operation
557 1   * in the Restore Engine that performs the unmarking, this plug-in function
558 1   * must periodically check for user-signalled cancelation,
        *                                                       and update
559 1   * progress data using the progress callback function argument.
560 1   *
561 1   * NOTE: This functions is responsible for keeping the volumes needed
        *                                                       list
562 1   * (ebvllist) element of the restore context up to date.
563 1   *
564 1   * UnmarkObject Parameters:
565 1   *
566 1   * context       (I) - Pointer to the restore context
567 1   * thisObject    (I) - The restoral object;
        *                     can be a leaf object (e.g. a
568 1   *                     file), or a container object (
        *                     e.g., a directory).
569 1   * BadFilesOnly  (I) - allows unmarking ONLY of files of state BADDATA.
570 1   * descend       (I) - Should unmark operation descend to operate on the
        *                     content of container objects.
571 1   * BadFilesCount (O) - returns the file count with BADDATA.
572 1   * fileMarked    (O) - return the total files marked after this unmark
573 1   *
574 1   * lenMarkedFiles(O) - return the length of files marked after this unmark
575 1   * dirMarked     (O) - return the total directories marked after this unmark
576 1   * otherMarked   (O) - return the total "other" files marked after this unmark
577 1   * progressCB    (I) - pointer to callback function to report progress and
        *                     test for cancellation
578 1   *
579 1   ****************************************************************/
581 1  eerrno_ty RSTPI_UnmarkObject( restore_context                      *context,
582 1                               struct RSTRPC_user_restorable_object *thisObject,
583 1                               const boolean_ty                     BadFilesOnly,
584 1                               const boolean_ty                     descend,
585 1                               unsigned long                        *BadFilesCount,
586 1                               unsigned long                        *fileMarked,
587 1                               u_hyper                              *lenMarkedFiles,
588 1                               unsigned long                        *dirMarked,
589 1                               unsigned long                        *otherMarked,
```

```
590 1    RSTPI_MarkProgressProc progressCB );
591 1
592 1  /*****************************************************************
593 1   * Is Object Marked
594 1   *
595 1   * This function determines if a restorable object has been
        *                                                       marked for restoration.
596 1   * It is intended to allow the user to determine the
597 1   * current restore markings for the restorable objects at the same object tree
598 1   * hierarchy level, i.e. objects that have the same parent restorableObject.
599 1   *
600 1   * Parameters:
601 1   *
602 1   * context       (I) - Pointer to the restore context
603 1   * thisObject    (I) - The restoral object to be checked: can be a leaf object
604 1   *                     (e.g. a file), or a container object ( e.g., a directory).
605 1   * marked        (O) - boolean to receive the marked(1) / unmarked( 0) result
606 1   *
607 1   ****************************************************************/
608 1  eerrno_ty RSTPI_IsObjectMarked( restore_context                      *context,
609 1                                 struct RSTRPC_user_restorable_object *thisObject,
610 1                                 boolean_ty                           *marked );
611 1
612 1  /*****************************************************************
613 1   * Is Object Markable
614 1   *
615 1   * Function Description:
616 1   *   Returns TRUE if the specified object is markable by the user,
617 1   *   returns FALSE if it is not. This function applies only to
618 1   *   container (directory) and leaf (file) objects.
619 1   *
620 1   * Parameters:
621 1   *   context       - (I) Pointer to the restore context
622 1   *   thisObject    - (I) ptr to the restorableObject in question
623 1   *
624 1   * Return:
625 1   *   TRUE  - the specified object is markable by the user;
626 1   *   FALSE - the specified object is not markable by the user.
627 1   *
628 1   ****************************************************************/
629 1  boolean_ty RSTPI_IsObjectMarkable( restore_context                      *context,
630 1                                     struct RSTRPC_user_restorable_object *thisObject );
631 1
632 1  /*****************************************************************
633 1   * Get Necessary Media:
634 1   *
635 1   * This OPTIONAL function is provided to allow retrieval of the
636 1   * media necessary to restore the currently marked objects. If a plugin does
637 1   * not supply this function, then it must maintain the ebvllist volume list
638 1   * attached to the restore context whenever mark and unmark are called, so that
639 1   * the generic RSTSL_GetNecessaryMedia function can retrieve the media list.
640 1   *
641 1
642 1   *
```

```
643 1 *  Parameters:
644 1 *    context      (I) - Pointer to the restore context
645 1 *    objects      (
646 1 *                 O) - address to return a pointer to the list of objects in
647 1 *   numberEntries (
648 1 *                 O) - the real number of media objects returned in the array
649 1 eerrno_ty RSTPI_GetNecessaryMedia( restore_context *context,
650 1                                    struct RSTRPC_media_list **objects,
651 1                                    short *numberEntries );
653 1 /*********************************************************************
654 1 *
655 1 * Get All Backup Times
656 1 *
657 1 * Function Description:
658 1 *    Retrieve the dates of the backups within the time range
659 1 *    specified by the caller.
660 1 *
661 1 * Parameters:
662 1 *    context    (I) - Pointer to the restore context
663 1 *    startTime  (I) - include no earlier than this date
664 1 *    endTime    (I) - include no later than this date
665 1 *    flags      (I) - Backup constraint flags; e.g. full-only/partial-ok
666 1 *    timesList  (O) - ptr to linked list of times
667 1 *    numEntries (O) - count of times returned
668 1 *
669 1 * Return Codes:
670 1 *    E_SUCCESS  - operation completed successfully
671 1 *
672 1 *********************************************************************/
674 1 eerrno_ty
675 1 RSTPI_GetAllBackupTimes( restore_context *context,
676 1                          const time_t startTime,
677 1                          const time_t endTime,
678 1                          RSTRPC_backup_flags_ty flags,
679 1                          struct RSTRPC_time_list **timesList,
680 1                          short *numEntries );
682 1 /*********************************************************************
683 1 *
684 1 * RSTPI_GetCurrentBackupTime
685 1 *
686 1 * Function Description:
687 1 *    Retrieve the time of the backup that the current restore
688 1 *    context is set to and return it in the preallocated buffer.
689 1 *
690 1 * Parameters:
691 1 *    context   (I) - Pointer to the restore context
692 1 *    bkupTime  (O) - the time of the backup
693 1 *
694 1 * Return Codes:
695 1 *    E_SUCCESS                     - operation completed successfully
696 1 *    EP_RB_RECOVER_INVALOP         - call issued out of sequence
697 1 *    EP_RB_RECOVER_BAD_ARGS        - invalid input argument
698 1 *    EP_RB_RECOVER_NO_CURR_BACKUP  - no valid backup currently
700 1 *********************************************************************
702 1 eerrno_ty
```

```
703 1 RSTPI_GetCurrentBackupTime( restore_context *context,
704 1                             time_t *bkupTime );
706 1 /*********************************************************************
707 1 *
708 1 * Set Backup For Time
709 1 *
710 1 * Function Description:
711 1 *    Switch to the backup plane of the specified time,
712 1 *    that is before the specified time,
713 1 *                                 or the most recent
714 1 *                                 if an exact match is not possible.
715 1 * Parameters:
716 1 *    context  (I) - Pointer to the restore context
717 1 *    forTime  (I) - The time for which the backup is requested
718 1 *    flags    (I) - Backup constraint flags: e.g., full-only/partial-ok
719 1 *
720 1 * Return Codes:
721 1 *    E_SUCCESS          - operation completed successfully
722 1 *    EP_RB_RECOVER_xxx  - backup plane cannot be found
723 1 *
725 1 *********************************************************************/
726 1 eerrno_ty
727 1 RSTPI_SetBackupForTime( restore_context *context,
728 1                         const time_t forTime,
      1                         RSTRPC_backup_flags_ty flags );
730 1 /*********************************************************************
731 1 *
732 1 * Set Previous Backup
733 1 *
734 1 * Function Description:
735 1 *    Set the restore context to that of the previous backup with
736 1 *    respect to the current one.
737 1 *
738 1 * Parameters:
739 1 *    context (I) - Pointer to the restore context
740 1 *    flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
741 1 *
742 1 * Return Codes:
743 1 *    E_SUCCESS                - operation completed successfully
744 1 *    EP_RB_NO_PREV_CATALOG    - when at the first catalog
745 1 *    EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
746 1 *                             file of the new catalog
747 1 *
748 1 *********************************************************************/
750 1 eerrno_ty
751 1 RSTPI_SetPrevBackup( restore_context *context,
752 1                      RSTRPC_backup_flags_ty flags );
754 1 /*********************************************************************
755 1 *
756 1 * Set Next Backup
757 1 *
758 1 * Function Description:
759 1 *    This routine must set the restore environment to the the next
760 1 *    of the current top level object.    backup
```

```
761  1  *  Parameters:
762  1  *     context   (I) - Pointer to the restore context
763  1  *     flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
764  1  *
765  1  *  Return Codes:
766  1  *     E_SUCCESS          - operation completed successfully
767  1  *     EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent
768  1  *                                     catalog
769  1  *     EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                               file
770  1  *                                       of the new catalog
771  1  *     EP_RB_RECOVER_NO_CATALOG - when mcat_set_mcplane failed
772  1  *
773  1  ***********************************************************************/
775  1  eerrno_ty
776  1  RSTPI_SetNextBackup( restore_context        *context,
777  1                       RSTRPC_backup_flags_ty    flags );
779  1  /**********************************************************************
781  1  *
782  1  *  Set First Backup
783  1  *
784  1  *  Function Description:
785  1  *     Set the restore context to that of the first backup plane.
786  1  *
787  1  *  Parameters:
788  1  *     context   (I) - Pointer to the restore context
789  1  *     flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
790  1  *
791  1  *  Return Codes:
792  1  *     E_SUCCESS          - operation completed successfully
793  1  *     EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                               file of
794  1  *                                       the new catalog
795  1  *
796  1  **********************************************************************/
798  1  eerrno_ty
799  1  RSTPI_SetFirstBackup( restore_context        *context,
800  1                        RSTRPC_backup_flags_ty    flags );
802  1  /**********************************************************************
803  1  *
804  1  *  Set Most Recent Backup
805  1  *
806  1  *  Function Description:
807  1  *     Set the restore context to that of the most recent backup
808  1  *     plane.
809  1  *
810  1  *  Parameters:
811  1  *     context   (I) - Pointer to the restore context
812  1  *     flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
813  1  *
814  1  *  Return Codes:
815  1  *     E_SUCCESS          - operation completed successfully
816  1  *     EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
817  1  *                                       file of
```

---

```
818  1  *                                     the new catalog.
819  1  *
820  1  **********************************************************************/
822  1  eerrno_ty
823  1  RSTPI_SetMostRecentBackup( restore_context        *context,
824  1                             RSTRPC_backup_flags_ty    flags );
826  1  /**********************************************************************
827  1  *
828  1  *  Is There Prev Backup
829  1  *
830  1  *  Function Description:
831  1  *     Determine if a backup exists prior to the backup that is
832  1  *     currently selected.
833  1  *
834  1  *  Parameters:
835  1  *     context   (I) - Pointer to the restore context
836  1  *     flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
837  1  *     isThere   (O) - TRUE/FALSE that requested backup does exist
838  1  *
839  1  *  Return Codes:
840  1  *     E_SUCCESS          - operation completed successfully
841  1  *     EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
842  1  *
844  1  **********************************************************************/
845  1  eerrno_ty
846  1  RSTPI_IsTherePrevBackup( restore_context        *context,
847  1                           RSTRPC_backup_flags_ty    flags,
                                 boolean_ty                *isThere );
849  1  /**********************************************************************
850  1  *
851  1  *  Is There Next Backup
852  1  *
853  1  *  Function Description:
854  1  *     Determine if a backup exists after the backup that is
855  1  *     currently selected.
856  1  *
857  1  *  Parameters:
858  1  *     context   (I) - Pointer to the restore context
859  1  *     flags     (I) - Backup constraint flags: e.g., full-only/partial-ok
860  1  *     isThere   (O) - TRUE/FALSE that requested backup does exist
861  1  *
862  1  *  Return Codes:
863  1  *     E_SUCCESS          - operation completed successfully
864  1  *     EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
865  1  *
867  1  **********************************************************************/
868  1  eerrno_ty
869  1  RSTPI_IsThereNextBackup( restore_context        *context,
870  1                           RSTRPC_backup_flags_ty    flags,
                                 boolean_ty                *isThere );
872  1  /**********************************************************************
873  1  *  Is There Prev Backup For Time
```

```
 874  1  *
 875  1  * Function Description:
 876  1  *     Determine if a backup exists prior to the specified time
 877  1  *
 878  1  *
 879  1  * Parameters:
 880  1  *     context (I) - Pointer to the restore context
 881  1  *     thisTime(I) - Time for the query
 882  1  *     flags  (
 883  1  *
 884  1  *          I) - Backup constraint flags: e.g., full-only/partial-ok
 885  1  *     isThere (O) - TRUE/FALSE that requested backup does exist
 886  1  *
 887  1  * Return Codes:
 888  1  *     E_SUCCESS          - operation completed successfully
      1  *     EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
      1  *
      1  ****************************************************************
      1  */
 890  1  eerrno_ty
 891  1  RSTPI_IsTherePrevBackupForTime( restore_context        *context,
 892  1                                  const time_t            thisTime,
 893  1                                  RSTRPC_backup_flags_ty  flags,
 894  1                                  boolean_ty             *isThere );
 896  1  /****************************************************************
 897  1  * Is There NextBackup For Time
 898  1  *
 899  1  * Function Description:
 900  1  *     Determine if a backup exists after to the specified time
 901  1  *
 902  1  * Parameters:
 903  1  *     context (I) - Pointer to the restore context
 904  1  *     thisTime(I) - Time for the query
 905  1  *     flags  (
 906  1  *          I) - Backup constraint flags: e.g., full-only/partial-ok
 907  1  *     isThere (O) - TRUE/FALSE that requested backup does exist
 908  1  *
 909  1  * Return Codes:
 910  1  *     E_SUCCESS          - operation completed successfully
 911  1  *     EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
 912  1  ****************************************************************
      1  */
 914  1  eerrno_ty
 915  1  RSTPI_IsThereNextBackupForTime( restore_context        *context,
 916  1                                  const time_t            thisTime,
 917  1                                  RSTRPC_backup_flags_ty  flags,
 918  1                                  boolean_ty             *isThere );
 920  1  #ifdef __cplusplus
 921  1  }
 922  1  #endif
 924  1  #endif  /* #ifndef H_RSLPIAPI */
```